A type-sound calculus of computational fields

A Type-Sound Calculus of Computational Fields ☆

Ferruccio Damiani*,[a], Mirko Viroli[b], Jacob Beal[c]

[a] *University of Torino, Italy (ferruccio.damiani@unito.it)*
[b] *University of Bologna, Italy (mirko.viroli@unibo.it)*
[c] *Raytheon BBN Technologies, USA (jakebeal@bbn.com)*

Abstract

A number of recent works have investigated the notion of "computational fields" as a means of coordinating systems in distributed, dense and dynamic environments such as pervasive computing, sensor networks, and robot swarms. We introduce a minimal core calculus meant to capture the key ingredients of languages that make use of computational fields: functional composition of fields, functions over fields, evolution of fields over time, construction of fields of values from neighbors, and restriction of a field computation to a sub-region of the network. We formalise a notion of type soundness for the calculus that encompasses the concept of domain alignment, and present a sound static type inference system. This calculus and its type inference system can act as a core for actual implementation of coordination languages and models, as well as to pave the way towards formal analysis of properties concerning expressiveness, self-stabilisation, topology independence, and relationships with the continuous space-time semantics of spatial computations.

Key words: Computational field, Core calculus, Operational semantics, Spatial computing, Type inference system, Type soundness

1. Introduction

In a world ever more densely saturated with computing devices, it is increasingly important to have effective tools for developing coordination strategies that can govern collections of these devices [8]. The goals of such systems are typically best expressed in terms of operations and behaviours over aggregates of devices, e.g., "send a tornado warning to all phones in the forecast area," or "activate all displays guiding me along a route towards the nearest group of my friends." The available models and programming languages for constructing distributed systems, however, have generally operated at the level of individual devices and their interactions, thereby obfuscating the design process. Effective models and programming languages are needed to allow the construction of distributed systems at the natural level of aggregates of devices. These must also be associated with a global-to-local mapping that links the aggregate-level specification to the operations and interactions of individual devices that are necessary to implement it.

Recently, approaches based on models of computation over continuous space and time have been introduced, which promise to deliver aggregate programming capabilities for the broad class of *spatial computers* [10]: networks of devices embedded in space, such that the difficulty of moving information between devices is strongly correlated

☆This work has been partially supported by project HyVar (www.hyvar-project.eu, this project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644298 - Damiani), by EU FP7 project SAPERE (www.sapere-project.eu, under contract No 256873 - Viroli), by ICT COST Action IC1402 ARVI (www.cost-arvi.eu - Damiani), by ICT COST Action IC1201 BETTY (www.behavioural-types.eu - Damiani), by the Italian MIUR PRIN 2010/2011 2010LHT4KM project CINA (sysma.imtlucca.it/cina - Damiani & Viroli), by Ateneo/CSP project RunVar (Damiani), and by the United States Air Force and the Defense Advanced Research Projects Agency under Contract No. FA8750-10-C-0242 (Beal). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings contained in this article are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release; distribution is unlimited.

*Corresponding Author.

with the physical distance between devices. Examples of spatial computers include sensor networks, robot swarms, mobile ad-hoc networks, reconfigurable computing, emerging pervasive computing scenarios, and colonies of engineered biological cells.

A large number of formal models, programming languages, and infrastructures have been created with the aim of supporting computation over space-time, surveyed in [7]. Several of these are directly related to the field of coordination models and languages, such as the pioneer model of TOTA [29], the (bio)chemical tuple-space model [47], the $\sigma\tau$-Linda model [50], and the pervasive ecosystems model in [52]. Their recurrent core idea is that through a process of diffusion, recombination, and composition, information injected in one device (or a few devices) can produce global, dynamically evolving *computational fields*—functions mapping each device to a structured value. Such fields are aggregate-level distributed data structures which, due to the ongoing feedback loops that produce and maintain them, are generally robust to changes in the underlying topology (e.g., due to faults, mobility, or openness) and to unexpected interactions with the external environment. They are thus useful for implementing and composing self-organising coordination patterns to adaptively regulate the behaviour of complex distributed systems [29, 47, 48].

A sound engineering methodology for space-time coordination systems will require more than just specification, but also the ability to predict to a good extent the behaviour of computational fields from the underlying local interaction rules—a problem currently solved only for a few particular cases (e.g., [6, 3]). This paper contributes to that goal by:

1. Introducing the *computation field calculus (*CFC*)*, a minimal core calculus meant to precisely capture a set of key ingredients of programming languages supporting the creation of computational fields: composition of fields, functions over fields, evolution of fields over time, construction of fields of values from neighbours, and restriction of a field computation to a sub-region of the network.

2. Formalising a notion of type soundness for CFC that encompasses the concept of *domain alignment* (i.e., proper sharing of information between devices), and presenting a sound static type inference system supporting polymorphism à la ML [17]. The main challenges in the design of the type system are to ensure domain alignment (which is complicated by the fact that the same expression may be evaluated many times, even recursively) and to support polymorphism à la ML without breaking domain alignment.

CFC is largely inspired by Proto [5, 33], the archetypal spatial computing language (and is in fact a much simpler fragment of it). As with Proto, it is based on the idea of expressing aggregate system behaviour by a functional composition of operators that manipulate (evolve, combine, restrict) continuous fields. Critically, these specifications can be also interpreted as local rules on individual devices, which are iteratively executed in asynchronous "computation rounds", comprising reception of all messages from neighbours, computing the local value of fields, and spreading messages to neighbours. The operational semantics of the proposed calculus precisely models single device computation, which is ultimately responsible for all execution in the network. The distinguished interaction model of this approach, which is formalised into a calculus in this paper, is based on representing state and message content in an unified way as an annotated evaluation tree. Field construction, propagation, and restriction are then supported by local evaluation "against" the evaluation trees received from neighbours. Not only is field calculus much simpler than Proto (and thus a tractable target for analysis), but the proposed formalisation also goes beyond Proto (which is a dynamically typed language) by introducing a static type inference system and a type soundness property that encompasses the notion of domain alignment, thereby enabling static analysis of the soundness and resilience properties of field computations.

The work thus developed formalises key constructs of existing coordination languages or models targeting spatial computing. As such, we believe that the calculus and its type inference system pave the way towards formal analysis of key properties applicable to various coordination systems, concerning expressiveness, self-stabilisation, topology independence, and relationships with the continuous space-time semantics of spatial computations.

The remainder of the paper is organised as follows: Section 2 describes the linguistic constructs of CFC and their application to system coordination. Section 3 illustrates how single devices interpret the CFC constructs locally. Section 4 illustrates some examples of programs to show the expressiveness of CFC. Section 5 formalises the operational semantics of CFC. Section 6 illustrates some examples of ill-formed programs to motivate the design of the type system. Section 7 presents the type inference system and its key properties (domain alignment and type soundness). Section 8 briefly surveys the main elements of a toolchain that is under development and grounds on CFC. Finally,

e	::=	x | l | (o $\overline{\text{e}}$) | (d $\overline{\text{e}}$) | (rep x w e) | (nbr e) | (if e e e)		expression
l	::=	n | b | $\langle$l,l$\rangle$		local value
w	::=	x | l		variable or local value
D	::=	(def d($\overline{\text{x}}$) e)		user-defined function declaration
P	::=	$\overline{\text{D}}$ e		program

Figure 1: CFC surface syntax

Section 9 discusses related work and Section 10 concludes by outlining possible directions for future work. The appendix contains the proofs of the main results.

This manuscript is an extended version of the prior [49] with more thorough discussions throughout, a detailed description of examples, a sound type inference system, the formal presentation and the proofs of the main results, and discussion of their application in tool construction.[1]

2. Computational field mechanisms

Generalising the common notions of scalar and vector field in physics, a computational field is a map from every computational device in a space to an arbitrary computational object. Examples of fields used in distributed situated systems include temperature in a building as perceived by a sensor network (a scalar field), the best routes to get to a location (a vector field), the area near an object of interest (a Boolean indicator field), or the people allowed access to computational resources in particular areas (a set-valued field). With careful choice of operators for manipulating fields, the aggregate and local views of a program can be kept coherent and each element of the aggregate-level program can be implemented by simple, automatically generated local interaction rules [4]. Following this idea, in this section we present a language to express such programs, identified based on the strengths and commonalities across many different approaches to spatial computing reviewed in [7] (though we do not rule out the possibility that others may be identified), and drawing particularly on the Proto [5, 33] implementations of these mechanisms.

We describe the selected mechanisms directly using the surface syntax of CFC, reported in Figure 1.[2] We take the global, aggregate-level viewpoint, considering the main syntactic element e as being a field expression, or simply a field. Conceptually, evaluation of a field expression has hence to be seen as executed by the whole "computing machine" formed by the entire set of networked devices. Practically, each device will work in aysnchronous rounds, each time properly interpreting the whole field expression as will be detailed in the operational semantics given in following sections. As a standard syntactic notation in calculi for object-oriented and functional languages [26], we use the overbar notation to denote metavariables over lists, e.g., we let $\overline{\text{e}}$ range over lists of expressions, written $\text{e}_1\ \text{e}_2\ \ldots\ \text{e}_n$.

A basic expression can be a literal value l (also called local value), such as a floating point number, a Boolean, or a pair—note most of the ideas of computational fields are agnostic to the structure of such values. According to the global viewpoint, a literal field expression l actually represents the constant function mapping l to all nodes. A basic expression can also be a variable x, which can be the formal parameter of a function or a store of information to support stateful computations (see rep construct below).

Such basic expressions (values and variables) can be composed by the following five constructs. The first one is *functional composition*, a natural means of manipulating fields as they are functions themselves: $(\text{o}\ \text{e}_1\ \text{e}_2\ \ldots\ \text{e}_n)$ is the field obtained by composing together all the fields e_1, e_2, ..., e_n by a built-in operator o (simply called operator). Operators include standard mathematical ones (e.g. addition, sine), as well as sensors, actuators, and others described below. Such operators are applied in a pointwise manner to all devices. For instance, if e_t is a field of Fahrenheit temperatures, then the corresponding field of Celsius temperatures is naturally written (* (/ 5 9) (- e_t 32)). Execution of built-in operators is context-dependent, i.e., it can be affected by the current state of the external world.

[1] In particular, Sections 4, 7, 8 and Appendix are new, and the rest of the text is expanded and updated as needed.

[2] The syntax in Figure 1 is indeed a subset of the syntax of the Proto language. Therefore, each CFC program is literally an executable Proto program. Furthermore, no expressiveness is lost: field calculus only omits syntactic sugar and relegates a large collection of assumed API, measurement, and summary functions to the semantically simple class of "built-in" operators. Any minor differences are due choices in which built-in operators to implement and how they are implemented in the Proto instantiation at hand.

For example, the 0-ary operator `uid` gives a field that maps each device to its unique device identifier, `dt` maps each device to the time elapsed since its previous computation round, and `nbr-range` produces a "field of fields" that maps each device to a table associating estimated distances to each neighbour (such a table being a field itself).

The second construct is *function (definition and) call*, which we use as abstraction tool and to support recursion: $(\texttt{d}\ \texttt{e}_1\ \texttt{e}_2\ \ldots\ \texttt{e}_n)$ is the field obtained as result of applying user-defined function `d` to the fields $\texttt{e}_1$, $\texttt{e}_2$, ... $\texttt{e}_n$. Such functions are declared with syntax $(\texttt{def}\ \texttt{d}(\overline{\texttt{x}})\ \texttt{e})$. For instance, given definition `(def convert (x) (* (/ 5 9) (- x 32)))`, expression `(convert` $\texttt{e}_t$`)` denotes the same field of Celsius temperatures as above. Note that function definitions, along with the top-level expression, form a program `P`.

The third construct is *time evolution*, used to keep track of a changing state over time: $(\texttt{rep}\ \texttt{x}\ \texttt{w}\ \texttt{e})$ is initially the field `w` (a local value or a variable) that is stored in the new variable `x`, and at each step in time is updated to a new field as computed by `e`, based on the prior value of `x`. For instance, `(rep x 0 (+ x 1))` is the (evolving) field counting in each device how many rounds that device has computed. Similarly, `(rep x 0 (+ x (dt)))` is the field of time passing, updated in each node as each new round is computed there.

The fourth construct is *neighbourhood field construction*, the mechanism by which information moves between devices: $(\texttt{nbr}\ \texttt{e})$ observes the value of `e` across neighbours, producing a "field of fields" like for the result of `nbr-range` described above. In particular, each device is mapped to a table associating each of its neighbors to that neighbour's value of `e`. As an example, consider three devices $\delta_1, \delta_2, \delta_3$, on which field `e` holds values 4, 5 and 6 respectively, and that the neighbourhood relation is the least symmetric and reflexive relation in which δ_1, δ_2 and δ_2, δ_3 are neighbours. In this case $(\texttt{nbr}\ \texttt{e})$ is a field mapping δ_1 to $(\delta_1 \mapsto 4, \delta_2 \mapsto 5)$, δ_2 to $(\delta_1 \mapsto 4, \delta_2 \mapsto 5, \delta_3 \mapsto 6)$ and δ_3 to $(\delta_2 \mapsto 5, \delta_3 \mapsto 6)$. So, letting `min-hood` be the built-in operator that takes a neighbourhood field and returns its minimum value, then `(min-hood (nbr e))` is the field mapping each device to the minimum value perceived in its neighbourhood, which would in this case be $(\delta_1 \mapsto 4, \delta_2 \mapsto 4, \delta_3 \mapsto 5)$.

The last construct is *domain restriction*, a sort of distributed branch: $(\texttt{if}\ \texttt{e}_0\ \texttt{e}_1\ \texttt{e}_2)$ is the field obtained by superimposing field $\texttt{e}_1$ computed everywhere $\texttt{e}_0$ is true and $\texttt{e}_2$ everywhere $\texttt{e}_0$ is false. As an example `(if` $\texttt{e}_{fah}$ $\texttt{e}_t$ `(convert` $\texttt{e}_t$`))` is the field of temperatures provided in Fahrenheit where the field $\texttt{e}_{fah}$ is true (i.e., in the subdomain in which it maps to true) and in Celsius everywhere else. Restriction is the most subtle of the five mechanisms, because it has the effect of preventing the unexpected spreading of computation to devices outside of the domain to which a computation has been restricted, even within arbitrarily nested function calls, as will be clarified in the following examples.

The CFC calculus is equipped with a type inference system which builds on the Hindley-Milner type system [17] for ML-like functional languages. The rules of the type system (presented in Section 7) specify a type inference algorithm (which is variant of the Hindley-Milner type inference algorithm [17]) that, given a function either fails (if the function cannon be typed) or returns its *principal type*, i.e., a type such that all the types that can be assigned to the function by the type inference rules can be obtained from the principal type by substituting:

- *local type variables* (ranged over by β) with *local types* (like the type of Booleans `bool`, the type of numbers `num`, and the types for pair values $\langle \texttt{bool}, \texttt{num} \rangle$, $\langle \langle \texttt{num}, \texttt{num} \rangle, \texttt{bool} \rangle$,...); and
- *type variables* (ranged over by α) with *types* (i.e., either local types or *field types* of the form $\texttt{field}(L)$ where L is a local type).

We now present some examples for the sake of illustrating how these five key mechanisms can be combined to implement useful spatial patterns. In the code of the examples presented through the paper, we use syntax coloring to increase readability: grey for comments, red for field calculus keywords, blue for user-defined functions, and green for built-in operators. The principal type of each function presented of the examples (as it would be inferred by the type inference algorithm specified in Section 7) is given in the comment inserted after the list of the formal parameters. We also use the following naming conventions for built-in operators: functions `*-hood` yield a local value `l` obtained by aggregating over the field value ϕ in input (e.g., `min-hood` of principal type $(\texttt{field}(\beta)) \to \beta$ returns the minimum among all values in each neighbourhood); and functions `pair` (of principal type $(\beta_1, \beta_2) \to \langle \beta_1, \beta_2 \rangle$), `fst` (of principal type $(\langle \beta_1, \beta_2 \rangle) \to \beta_1$), and `snd` (of principal type $(\langle \beta_1, \beta_2 \rangle) \to \beta_2$) respectively create a pair of local values and access a pair's first and second component. Additionally, given a built-in operator `o` that takes $n \geq 1$ locals an returns a local, the built-in operators `o[*,...,*]` are variants of `o` where one or more inputs are fields (as indicated in the bracket, `l` for local or `f` for field), and the return value is a field, obtained by applying operator `o` in a point-wise manner. For

instance, as = (of principal type $(\beta,\beta) \to \texttt{bool}$) compares two locals returning a Boolean, =[f,f] (of principal type $(\texttt{field}(\beta),\texttt{field}(\beta)) \to \texttt{field}(\texttt{bool})$) is the operator taking two field inputs and returns a Boolean field where each element is the comparison of the corresponding elements in the inputs, and similarly =[f,l] (of principal type $(\texttt{field}(\beta),\beta) \to \texttt{field}(\texttt{bool})$) takes a field and a local and returns a Boolean field where each element is the comparison of the corresponding element of the field in input with the local.

```
(def gossip-min (source)    ;; has type: (β) → β
  (rep d source (min-hood (nbr d))))

(def distance-to (source)    ;; has type: (bool) → num
  (rep d infinity (mux source 0 (min-hood (+[f,f] (nbr d) (nbr-range))))))

(def distance-obs-to (source obstacle)    ;; has type: (bool, bool) → num
     (if (not obstacle) (distance-to source) infinity))
```

We first exemplify how constructs **rep** and **nbr** can be nested to create a long-distance computation, to achieve network-wide propagation processes. Function gossip-min takes a source field (the formal parameter source has type β) and produces a new field mapping each device to the minimum value that source initially takes. The **rep** construct initially sets the output variable d (of type β) at source (of type β), and then iteratively updates the value at each device with the minimum d available at any neighbour. Hence, gossip-min describes a process of gossiping values until the minimum one converges throughout the network.

Similarly, function distance-to takes as its input a source field holding Boolean values (the formal parameter source has type bool), and returns a new numerical field that maps each device to the estimated distance to the nearest source node (function distance-to has return type num), i.e., to the nearest device where source is true. This works as follows:

- d (of type num) is initially set to infinity in each node;
- operator mux (of principal type $(\texttt{bool},\beta,\beta) \to \beta$) is a purely functional multiplexer (it computes all three inputs, then uses the first input to choose whether to return the second or third);
- sources are set to distance 0;
- other devices use the triangle inequality: we retrieve the sum (by the operator +[f,f] of principal type $(\texttt{field}(\texttt{num}),\texttt{field}(\texttt{num})) \to \texttt{field}(\texttt{num})$) of neighbour's distance value (**nbr** d) (of type $\texttt{field}(\texttt{num})$) and the corresponding neighbour's estimated distance (by the operator nbr-range of principal type $() \to \texttt{field}(\texttt{num})$), and then take the minimum (by the operator min-hood) of the resulting values.

The field returned by distance-to is often also referred to as a *gradient* [29, 6, 47], and is a key building block for many computations in mobile ad-hoc networks, such as finding routes to points of interest. There are many similar variants with different purposes, most of which automatically repair themselves when either the sources or network structure change.

The last definition exemplifies the use of construct **if**. It creates two different spatial domains: one where the obstacles are located (field obstacle holds positive Boolean value) and one where they are not. In the former a constant field with value infinity is computed; in the latter we compute the distance-to field. Since distance-to is being computed in a restricted spatial domain, information cannot be spread via **nbr** through areas where there are obstacles (semantics detailing the interaction of **nbr** and **if** are provided in Section 5). Hence, even though distance-to is implemented without any awareness of obstacles, the computation is prevented from spreading through areas where it should not, and the distance estimation as provided by distance-to is modulated in order to take into account the need of circumventing obstacle areas.

Note that **if** works like a mux executed in a network where neighbouring relations between nodes that took different branches are disabled—in the example above, it would be like separating the obstacle region from the rest of the network.

A number of coordination mechanisms can be constructed on the basis of the above examples, like the gradient-based patterns discussed in [47, 52, 50], which find applications in many areas, including crowd steering in pervasive computing. Examples of implementation of several of these coordination mechanisms are illustrated in Section 4.

3. From global to local viewpoint

The description of field constructs so far has focused on what we can call the *global viewpoint*, in which the computation is considered as occurring on the overall computational fields distributed in the network. For the calculus to be actually executed, however, each device has to perform a specific set of actions at particular times, including interaction with neighbours and local computations. The result of these local actions then produces the overall evolution of computational fields. We call this description of the language in term of individual devices the *local viewpoint*, and it is this view that we shall use for the operational semantics. Let us now begin with an informal presentation of the peculiar aspects of that operational semantics, to aid in understanding the full formalisation presented in Section 5.

Following the approach considered in Proto [33] and many other distributed programming languages, devices undergo computation in rounds. In each round, a device sleeps for some limited time, wakes up, gathers information about messages received while sleeping, performs its actual field evaluation, and finally emits a message to all neighbours with information about the outcome of computation, before going back to sleep.

Taking the local viewpoint, we may model a field computation by modeling the evaluation of a single device at a single round, assuming the scheduling of such rounds across the network to be fair and non-synchronous—either fully asynchronous or partially synchronous, meaning that devices cannot execute infinitely quickly. Assuming that the main bottleneck in the system is communication rather than computation (which is frequently the case in wireless communication networks), this model can be readily achieved by any collection of devices with internal clocks that schedule execution of rounds at regular intervals. So long as the relative drift between clocks is bounded, execution on such a system will be fair and partially synchronous.

To support the combination of field constructs, we design our operational semantics as follows. First, our functional style of composition, definition, and calls fits well with a small-step evaluation semantics, in which we start from the initial expression to evaluate and reduce it to a normal form representing the outcome of computation, including the local value of the resulting field and the information to be spread to neighbours. In order to keep track of the state of variables introduced by `rep` constructs, and values at `nbr` constructs to be exchanged with neighbours, we take our computational state to be a dynamically produced evaluation tree. During a round of computation, such a tree is incrementally decorated with partial results expressed as *annotations* of the form "$\cdot v$" or *superscripts* "$^{\mathtt{s}}$". These decorations track the local outcome of evaluation and determine which subexpression will be next evaluated.

To illustrate our management of evaluation order and computational rounds, as well as the `rep` construct, let us begin by considering expression `(rep x 0 (+ x 1))` of type `num` (cf. Section 2). As this tree is evaluated according to the operational semantics, it goes through a sequence of four *top-level* steps—a top level step corresponds to a rule applied to the whole tree. A top-level step may involve *nested* steps—a nested step corresponds to a rule applied to a subtree. We show these top-level steps informally by underlining in each top-level step the next portion of the tree to be rewritten, by colouring the changes introduced by each rewrite in red,[3] and by labelling each top-level step with the names of rules of the operational semantics causing the top-level step and its nested steps. The rules may be ignored at a first reading, and be considered later to understand the formal calculus in Section 5. The first computation round goes as follows:

$$\texttt{(rep x 0 (+ }\underline{\texttt{x}}\texttt{ 1))} \xrightarrow{\text{[REP,CONG,VAR]}} \texttt{(rep x 0 (+ x}\cdot\texttt{0 }\underline{\texttt{1}}\texttt{))} \xrightarrow{\text{[REP,CONG,VAL]}} \underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1))}} \xrightarrow{\text{[REP,CONG,OP]}}$$
$$\underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}} \xrightarrow{\text{[REP]}} \texttt{(rep}^{1}\ \texttt{x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}\cdot\texttt{1}$$

Annotations are computed depth-first in the expression tree until eventually reaching the outer expression: we first annotate variable `x` with its current (initial) value 0, then simply identically annotate value 1, then perform built-in operation `+` causing annotation of its sub-tree with 1, and finally execute the `rep` construct, which records the result value as a superscript to `rep` and as an annotation of the whole expression.

[3] They will appear grey in a non-colour print of the paper.

Once the evaluation is complete, with the result value in the outer-most annotation, the whole evaluation tree will be shipped as a message to neighbours, in order to support the semantics of `nbr` statements, which locally gather values stored as annotations in neighbour's evaluation trees. Pragmatically, of course, any implementation might massively compress the tree, sending only enough information to support the semantics of `nbr` statements.

The subsequent round begins after an initialization that erases all non-superscript decorations. This second round leads to evaluation tree $\texttt{(rep}^2\ \texttt{x 0 (+ x·1 1·1)·2)·2}$, the third one to $\texttt{(rep}^3\ \texttt{x 0 (+ x·2 1·1)·3)·3}$, and so on.

The main purpose of managing evaluation trees in this way is to support information exchange through the `nbr` construct. Consider the expression `(min-hood (nbr (t)))` of type `num` (cf. Section 2), where `t` of principal type $() \to \texttt{num}$ is a 0-ary built-in operator that returns the temperature perceived in each device. If a device δ perceives a temperature of 7 degrees Celsius, and executes its first computation round before its neighbours, then the result of computation should clearly be 7. This is implemented by the following sequence of transitions:

$$\texttt{(min-hood (nbr (t)))} \xrightarrow{\text{[CONG,CONG,OP]}} \texttt{(min-hood (nbr (t)·7))} \xrightarrow{\text{[CONG,NBR]}}$$
$$\texttt{(min-hood (nbr (t)·7)·}(\delta \mapsto 7)\texttt{)} \xrightarrow{\text{[OP]}} \texttt{(min-hood (nbr (t)·7)·}(\delta \mapsto 7)\texttt{)·7}$$

We first enter the subexpression with the 0-ary operator `t` which yields 7. We then evaluate `nbr` to the field of neighbour values, associating only δ to 7, written $(\delta \mapsto 7)$. Finally, we evaluate unary operator `min-hood`, which extracts the smallest element of the input field, which in this case is 7.

Construct `nbr` retrieves values from neighbours using the *tree environment* of the device δ, which models its store of recent messages received from neighbours. The tree environment is a mapping $\Theta = (\delta_1 \mapsto e_1, \ldots, \delta_n \mapsto e_n)$ created at each round, from neighbours (δ_i) to their last-received evaluation tree (e_i), which we call the *neighbour tree* of δ_i . The evaluation of $(\texttt{nbr}\ e)$, where e is evaluated to local value `l`, takes values from the tree environment to produce a field $(\delta \mapsto \texttt{l}, \delta_1 \mapsto \texttt{l}_1, \ldots, \delta_n \mapsto \texttt{l}_n)$, mapping δ to `l` and each neighbour δ_i to the corresponding local value $\texttt{l}_i$ from δ_i.

In the example above we assumed that none of the neighbours of δ had already completed a round of computation, and that therefore Θ was empty and accordingly `(nbr (t))` gave simply $(\delta \mapsto 7)$. If we instead assume that the first round of computation on the device δ takes place when the neighbours δ_1 and δ_2 have completed exactly one round of computation, perceiving temperatures of 4 and 9 degrees respectively, then the tree environment of δ would be $(\delta_1 \mapsto e_1, \delta_2 \mapsto e_2)$, where $e_1 = \texttt{(min-hood (nbr (t)·4)·}(\delta \mapsto 4)\texttt{)·4}$ and $e_2 = \texttt{(min-hood (nbr (t)·9)·}(\delta \mapsto 9)\texttt{)·9}$. The computation goes similarly, the only difference is that the evaluation of `(nbr (t)·7)` now produces the field $\phi = (\delta \mapsto 7, \delta_1 \mapsto 4, \delta_2 \mapsto 9)$ and the final outcome of the computation round on δ is the tree $\texttt{(min-hood (nbr (t)·7)·}\phi\texttt{)·4}$.

More specifically, the extraction of values from neighbours is achieved by computing the local evaluation tree "against" the set of its neighbour trees: when evaluation enters a subtree, in the tree environment Θ we correspondingly enter the corresponding subtree (when it matches) on all of its neighbour trees. Although each node executes the same program, the trees of two different nodes may not match. For instance, it may happen that on a node the left branch of an `if` expressions is evaluated, while on another node the right branch is evaluated (an example illustrating this situation is given in the last paragraph of this section). This process on neighbour trees is called *alignment*. So, in the example above, sub-tree `(nbr (t)·7)` is recursively evaluated against the neighbour sub-trees

$$(\delta_1 \mapsto \texttt{(nbr (t)·4)} \cdot (\delta_1 \mapsto 4), \delta_2 \mapsto \texttt{(nbr (t)·9)} \cdot (\delta_2 \mapsto 9))$$

in which the neighbour values are immediately available as the outermost annotation of the argument of `nbr`. So, the effect of an instance of `nbr` in a given position of the tree, is simply understood as gathering the annotations existing in the same position of all neighbours' evaluation trees—neighbours that actually have a value there are precisely called the aligned neighbours.

One reason for using this structural alignment mechanism is to seamlessly handle the cases where `nbr` subtrees could be nested at a deep level of the evaluation tree because of (possibly recursive) function calls. Assume the user-defined function definition `(def f (x) (min-hood (nbr x)))` of principal type $(\beta) \to \beta$, and the main expression `(f (t))` of type `num` (recall that the built-in operator `t` has principal type $() \to \texttt{num}$) whose expected behaviour is then equivalent to our prior example `(min-hood (nbr (t)))`. This expression would be handled by the following

sequence of transitions:

$$\texttt{(f (t))} \xrightarrow{\text{[CONG,OP]}} \texttt{(f (t)}\cdot 7\texttt{)} \xrightarrow{\text{[FUN,CONG,CONG,VAR]}} \texttt{(f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{))}}\ \texttt{(t)}\cdot 7\texttt{)} \xrightarrow{\text{[FUN,CONG,NBR]}}$$
$$\texttt{(f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot\phi\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)} \xrightarrow{\text{[FUN,OP]}} \texttt{(f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot\phi\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)}\cdot 4$$

After the function arguments are all evaluated, the second transition creates a superscript to function `f`, holding the evaluation tree corresponding to its body. This gets evaluated as usual, and its resulting annotation 4 is transferred to become the annotation of the function call. So, note that the evaluation tree is a dynamically expanding data structure because of such function superscripts being generated and navigated at each call, with alignment automatically handling `nbr` construct, even for arbitrary recursive function call structures. Note that this mechanism also prevents recursive calls from implying infinite evaluation trees, since only those calls that are actually made are annotated.

This management of memory trees also easily accommodates the semantics of restriction. An `if` subexpression is evaluated by first evaluating its condition, then evaluating the selected branch, and finally erasing all decorations on the non-taken branch, including superscripts. In this way, neighbour trees corresponding to devices that took a different branch will be automatically discarded at alignment time, since entering the same subexpression is impossible because of a bad match. For example, consider expression `(if (c) (f (t)) 0)`, where operator `c` of type $() \to$ `bool` returns a Boolean field that is true at δ and δ_2, and false at δ_1. Assuming again that first round of δ happens after first round of δ_1 and δ_2, we have:

$$\texttt{(if (c) (f (t)) 0)} \xrightarrow{\text{[CONG,OP]}} \texttt{(if (c)}\cdot\texttt{true (f (t)) 0)} \to^*$$
$$\texttt{(if (c)}\cdot\texttt{true (f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot(\delta\mapsto 7,\delta_2\mapsto 9)\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)}\cdot 7\ \texttt{0)} \xrightarrow{\text{[THEN]}}$$
$$\texttt{(if (c)}\cdot\texttt{true (f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot(\delta\mapsto 7,\delta_2\mapsto 9)\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)}\cdot 7\ \texttt{|0|)}\cdot 7$$

The reason why the `rep` subexpression now yields field $(\delta \mapsto 7, \delta_2 \mapsto 9)$ is that the neighbour tree of δ_1 cannot be aligned, for it has `(c)` annotated with `false`, which does not match. Hence, `nbr` will retrieve values only from the *aligned nodes* that followed the same branch, avoiding interference from nodes residing in different regions of the partition made by restriction. The erasure of the non-taken branch by operator $|.|$ (0 trivially erases to 0 in this case) is used to completely reinitialise computation there, since the node no longer belongs to the domain in which the non-taken branch should be evaluated.

4. Application to self-organisation

A notable application of spatial computing languages, and of CFC, is to implement *self-organising systems*. In this section we discuss various features of CFC with the goal of presenting its expressiveness, especially related to self-organizing systems. Such self-organising systems have the property that they create coherent system-level patterns out of the local interaction between individual system components. Often inspired by natural mechanisms and metaphors [28, 59], self-organisation provides a means to create systems which spontaneously manifest properties of adaptivity and robustness to environment changes of various kinds.

The CFC provides a terse means of expressing both individual self-organisation mechanisms and their composition to create robust and adaptive systems. Being able to express such mechanisms and systems using a well-suited minimal calculus will render them more amenable to mathematical investigation. Critically, we expect that this will enable the investigation of basis sets of self-organisation primitives, extending the approach in [23], and will enable investigation of how adaptivity and robustness are affected when self-organisation mechanisms interact.

We illustrate the efficacy of field-calculus for expressing self-organising systems with three examples of increasing complexity: Laplacian approximate consensus, adaptive distributed Voronoi partitioning, and a "channel" pattern for distributed route computation.

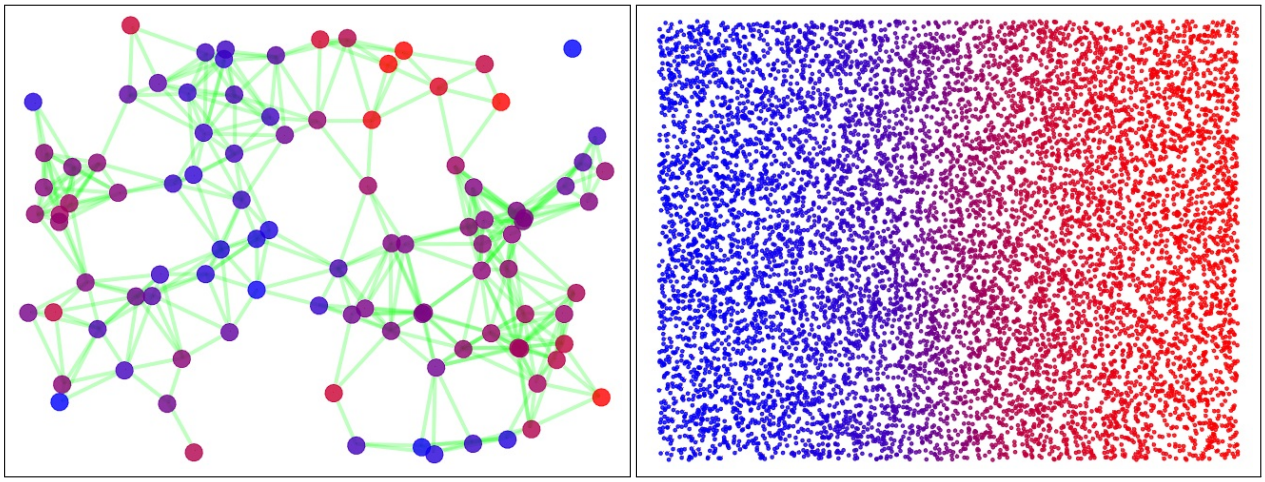

Figure 2: Laplacian consensus in a partially converged state, showing value on a color scale from 0 (blue) to 1 (red), deployed in a low-density network with initially random values and topology (green) in evidence (left), and from initially all 0 on the left half and all 1 on the right half in a high-density environment of 10,000 nodes (right). Simulation executed in MIT Proto [5, 33] on devices randomly distributed in 132x100 unit area with communication radius 15, run to quiescence.

Example 4.1 (Laplacian approximate consensus). Many distributed applications use approximate consensus as a building block for self-organising behavior, with examples spanning robot formation control [19], flocking and swarming [35], sensor fusion [56], modular robotics [57] and synchronisation [42]. Laplacian-based approaches are a common mechanism for distributed approximate consensus, in which each device finds a weighted local average of its own current value with the values held by its neighbours. It can be implemented in CFC as:

```
(def laplacian-consensus (init epsilon)    ;; has type: (num, num) → num
   (rep val init
      (+ val
         (* epsilon
            (sum-hood (-[f,l] (nbr val) val))))))
```

Each device tracks an estimated consensus value `val`, beginning with its own value `init`. At each round, this value is incrementally averaged with the current values held by neighbours, using `sum-hood` (of type $(\texttt{field}(\texttt{num})) \to \texttt{num}$) to sum the differences and multiplying by a small increment `epsilon`.

Figure 2 shows examples of Laplacian approximate consensus being computed. On the left, we show a low-density view over a mesh network with initially random values: in this case, the consensus mechanism converges very quickly. On the right, we apply the same field to a high-density network that more closely approximates continuous space, on which convergence is much slower, per [20].

Example 4.2 (Voronoi partitioning). Computing a Voronoi partition is an operation that is frequently useful in distributed systems. Given an initial set of "seed" devices, a Voronoi partition assigns each device to the partition of the nearest seed (by some not necessarily Euclidean metric), effectively breaking the network up into "zones of influence" around key elements. For example, in a mesh network with a few devices that serve as gateways to the more general external network, a distributed Voronoi partition can be used to identify which gateway each device should use for external communication. It can be realised in CFC as follows:

```
(def voronoi (seed id)    ;; has type: (bool, num) → num
  (snd
    (rep partition
         (pair infinity 0)
         (mux seed
              (pair 0 id)
              (min-hood (nbr (pair (distance-to seed) (snd partition))))))))
```

The Voronoi partition provides a simple example of creating a self-organising system by modulating (i.e., functionally combining) other self-organisation mechanisms, in this case the adaptive distance computation of the `distance-to` function.

The pattern begins with the `seed` input, a Boolean-valued field that is true for those devices that are seeds of the partition and false for all other devices (the formal parameter `seed` has type `bool`), and an `id` field associating each device with a unique identifier (the formal parameter `id` has type `num`). The `partition` variable (of type $\langle \texttt{num}, \texttt{num} \rangle$) then tracks a field of pairs (constructed by operator `pair` taking two numbers) containing the distance and identity for the nearest identifier. Seeds provide a base case, being distance zero from themselves. The rest of the partition is computed by lexicographic minimisation over the distance and identity pairs of each neighbour. Because this is based in a feed-forward manner on the self-stabilising `distance-to` function, this pattern too is self-stabilising, and will rapidly adapt to changes in the set of seeds or the structure of the network. Note operator `snd` extracts the second element of a pair, which is used at the top level to extract the `id` of the nearest seed.

Figure 3 shows examples of a Voronoi partition being computed. On the left, we show a low-density view over a mesh network, where it is essentially computing hop-count distances and produces a standard graph partition. On the right, we apply the same field to a high-density network that more closely approximates continuous space. Here the borders between regions in the network closely approximate the mathematical ideal of geometric shortest paths. This comparison is also an illustration of how field calculus can serve as a bridge between continuous aggregate semantics and discrete implementation.

Example 4.3 (The "channel' pattern). Finally, to show a more complex example of self-organising mechanism composition, we show a realisation of the so-called "channel" pattern (from [13]), which dynamically computes distributed routes between regions of a network, and dynamically adapts to shape and changes of the network topology. Examples of its use include long-range reliable communications, or advanced crowd steering applications in pervasive computing [58, 51].

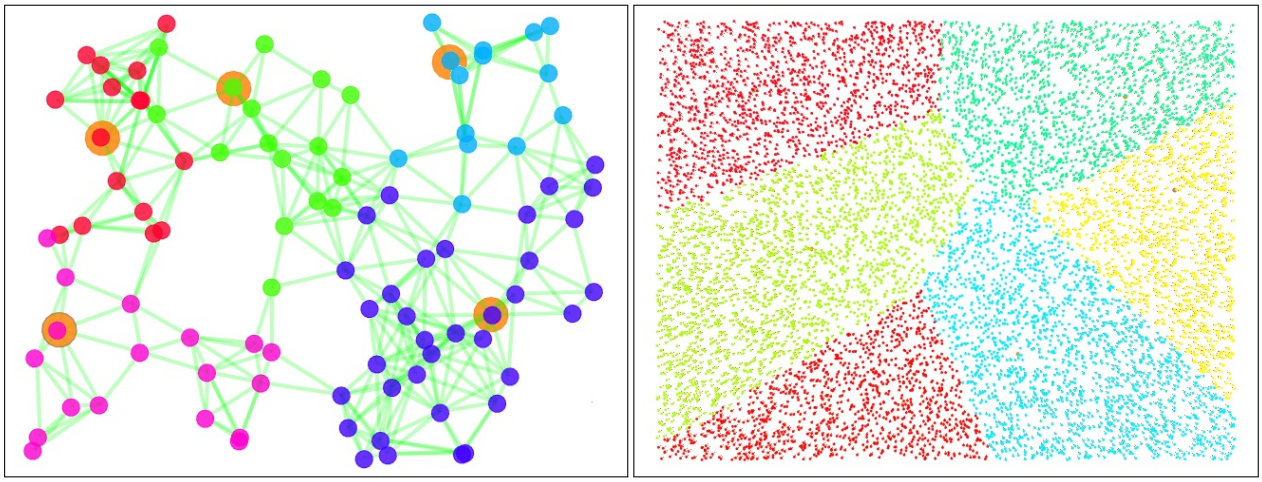

Figure 3: Voronoi partition from seeds (large orange circles), indicating partition by color, deployed in a low-density network with topology (green) in evidence (left), and in a high-density environment of 10,000 nodes (right). Simulation executed in MIT Proto [5, 33] on devices randomly distributed in 132x100 unit area with communication radius 15, run to quiescence.

The "channel" is a Boolean field that is true for devices near the shortest route from a given (distributed) source to a (distributed) destination. It can be realised as follows:

```
(def shortest-path (source destination)    ;; has type: (bool, bool) → num
  (rep path false (mux source true (any-hood (and[f,f]
      (nbr path)
      (=[l,f] (distance-to destination)
         (nbr (min-hood (nbr (distance-to destination))))))))))

(def channel (source destination width)    ;; has type: (bool, bool, num) → bool
   (< (distance-to (shortest-path source destination)) width))

(def channel-avoiding-obstacles (obstacle source destination width)
                ;; has type: (bool, bool, bool, num) → bool
    (if obstacle false (channel source destination width)))
```

The `shortest-path` function yields a field that is true only on devices in the shortest path between source and destination. It is true initially on source and false everywhere else, and a device δ becomes part of the path (i.e., it is mapped to true) if any of its neighbours (`any-hood` + `nbr`) is already in the `path`, and (by the built-in operator `and[f,f]` of principal type $(\texttt{field}(\texttt{bool}), \texttt{field}(\texttt{bool})) \rightarrow \texttt{field}(\texttt{bool})$) its neighbour with minimum distance to destination is as distant as δ—built-in operator `any-hood` (of principal type $(\texttt{field}(\texttt{bool})) \rightarrow \texttt{bool}$) yields true only if at least one of the elements of the field is true.

The `channel` function adds redundancy to a `shortest-path` route by using `distance-to` to select every device within the desired channel `width`. The `channel-avoiding-obstacles` function then modulates the whole complex of `channel`, `shortest-path`, and `distance-to` functions by using **if** to restrict the region where they are computed to that portion of the space without obstacles.

Figure 4 shows obstacle-avoiding channels being computed. On the left, we show a low-density view over a mesh network, in which the channel simply represents a point-to-point routing path. On the right, we apply the same field to a high-density network that more closely approximates continuous space: by applying the channel field to a distributed source, destination and obstacle we illustrate the way in which field computations can act as a bridge between continuous aggregate semantics and discrete implementation through local interaction rules. An example application of this structure is in the context of pervasive computing, to develop a steering service from people moving

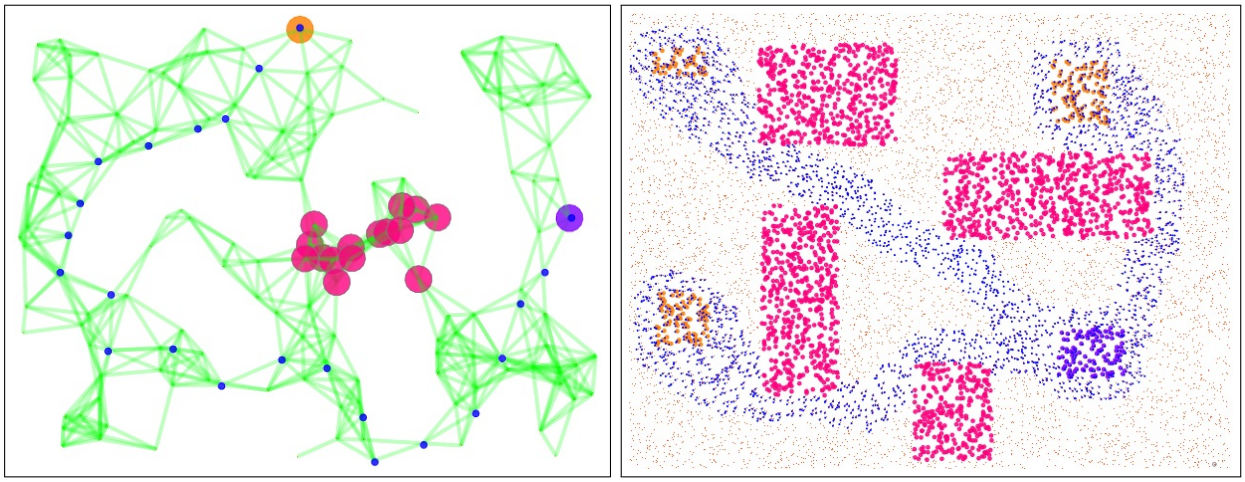

Figure 4: Channels (blue) route between source (orange) and destination (purple) around obstacles (pink), deployed in a low-density network with topology (green) in evidence (left), and in a high-density environment of 10,000 nodes (right). Simulation executed in MIT Proto [5, 33] on devices randomly distributed in 132x100 unit area with communication radius 15, run to quiescence.

in an articulated environment [47, 29]. The channel field can be used to enable all the devices (digital signs, peoples' smartphones, obstacle sensors, crowd detection sensors, etc.) that aid in guiding people moving from a source to a destination.

5. The computational field calculus

The surface syntax (Figure 1) and the mechanisms of CFC have been introduced and motivated in the previous sections. This section presents a formalisation of the CFC operational semantics. The various aspects of the formalisation are set forth in Figure 5, described here in turn after few preliminaries. We let δ range over device unique identifiers and ϕ over field values (mapping set of devices to local values). Given any meta-variable y we let $\mathring{y}$ range over an element y or the *null decoration* (which in the calculus is $\circ$ when it has to be expressed, and blank otherwise). The calculus is agnostic to the syntax of local values: we only assume they include at least device identifiers δ and Boolean values—denoted by f (for false) and t (for true).

5.1. Runtime expression syntax

A *runtime expression* is the evaluation tree created out of a surface expression. It is similar to expressions in the surface syntax (cf. Figure 1) with the following differences (see Figure 5):

(i) a (runtime) value v is either a local value $\mathtt{l}$ or a field value ϕ;

(ii) a run-time expression e can be coupled (at any level of depth) with optional *annotation* $\mathring{v}$ representing the transient side-effect of a computation;

(iii) constructs `rep` and function calls can have a *superscript* (`s`) representing the durable side-effect of a computation.

Note that, syntactically, surface syntax expressions can (and will) be used to denote runtime expressions with null decorations in all annotations and superscripts. Conversely, the erasure operator $|\cdot|$ is used to turn a runtime expression e (or an auxiliary rte a) to the surface expression $|e|$ (resp. $|a|$) obtained by dropping all annotations and superscripts.

A field value ϕ can either be written as $\delta_1 \mapsto \mathtt{l}_1, \ldots, \delta_n \mapsto \mathtt{l}_n$ or be shortened by notation $\overline{\delta} \mapsto \overline{\mathtt{l}}$. Note that fields are actually mappings, for which we introduce some syntactic conventions and operators. The domain of ϕ, which is the set $\{\delta_1, \ldots, \delta_n\}$, is denoted by $\mathbf{dom}(\phi)$. The value $\mathtt{l}_i$ associated to a given device δ_i by field ϕ is retrieved by notation $\phi(\delta_i)$. Since a field can be seen as a list, we use the notation $\bullet$ for the empty field, and comma as list concatenation operator: e.g. ϕ, ϕ' is the field having both the mappings of ϕ and ϕ'. We shall sometimes restrict

Runtime expression syntax:

e	$::=$	$a{\cdot}\mathring{v}$	runtime expression (rte)
a	$::=$	$\mathtt{x} \mid \mathtt{l} \mid (\mathtt{nbr}\, e) \mid (\mathtt{if}\, e\, e\, e) \mid (\mathtt{rep}^{\mathtt{s}}\, \mathtt{x}\, \mathtt{w}\, e) \mid (\mathtt{d}^{\mathtt{s}}\, \overline{e}) \mid (\mathtt{o}\, \overline{e})$	auxiliary rte
v	$::=$	$\mathtt{l} \mid \phi$	runtime value
$\mathtt{s}$	$::=$	$\mathring{a}$	superscript
$\mathtt{w}$	$::=$	$\mathtt{x} \mid \mathtt{l}$	variable or local value
ϕ	$::=$	$\overline{\delta} \mapsto \overline{\mathtt{l}}$	field value
Θ	$::=$	$\overline{\delta} \mapsto \overline{e}$	tree environment
Γ	$::=$	$\overline{\mathtt{x}} := \overline{v}$	variable environment

Congruence contexts:

$\mathbb{C} \quad ::= \quad (\mathtt{nbr}\ []) \mid (\mathtt{d}^{\mathtt{s}}\ \overline{e}\ []\ \overline{e}) \mid (\mathtt{o}\ \overline{e}\ []\ \overline{e}) \mid (\mathtt{if}\ []\ e\ e) \mid (\mathtt{if}\ a{\cdot}t\ []\ e) \mid (\mathtt{if}\ a{\cdot}f\ e\ []\ )$

Alignment contexts:

$\mathbb{A} \quad ::= \quad \mathbb{C} \mid (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ []) \mid (\mathtt{d}^{[]}\ \overline{a}{\cdot}\overline{v})$

Auxiliary functions:

$$\begin{array}{rcl} \pi_{\mathbb{A}}(\Theta, \Theta') & = & \pi_{\mathbb{A}}(\Theta), \pi_{\mathbb{A}}(\Theta') \\ \pi_{\mathbb{A}}(\delta \mapsto (\mathbb{A}'[e]){\cdot}v) & = & \delta \mapsto e \qquad \text{if } \mathbb{A}' :: \mathbb{A} \\ \pi_{\mathbb{A}}(\delta \mapsto e) & = & \bullet \qquad \text{otherwise} \end{array}$$

$$\mathtt{s} \rhd a = a \qquad \mathtt{s} \rhd \circ = \mathtt{s}$$

$$\begin{array}{rcl} (\mathtt{nbr}\ []) & :: & (\mathtt{nbr}\ []) \\ (\mathtt{d}^{\mathtt{s}'}\ e'_1 ... e'_{i-1}\ []\ e'_{i+1} ... e'_n) & :: & (\mathtt{d}^{\mathtt{s}}\ e_1 ... e_{i-1}\ []\ e_{i+1} ... e_n) \\ (\mathtt{o}\ e'_1 ... e'_{i-1}\ []\ e'_{i+1} ... e'_n) & :: & (\mathtt{o}\ e_1 ... e_{i-1}\ []\ e_{i+1} ... e_n) \\ (\mathtt{if}\ []\ e'_1\ e'_2)) & :: & (\mathtt{if}\ []\ e_1\ e_2)) \\ (\mathtt{if}\ a'{\cdot}t\ []\ e') & :: & (\mathtt{if}\ a{\cdot}t\ []\ e) \\ (\mathtt{if}\ a'{\cdot}f\ e'\ []\ ) & :: & (\mathtt{if}\ a{\cdot}f\ e\ []\ ) \\ (\mathtt{rep}^{\mathtt{s}'}\ \mathtt{x}\ \mathtt{w}\ []) & :: & (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ []) \\ (\mathtt{d}^{[]}\ e'_1 ... e'_n) & :: & (\mathtt{d}^{[]}\ e_1 ... e_n) \end{array}$$

Reduction rules:

$$\frac{}{\delta;\Theta;\Gamma \vdash (\mathtt{if}\ a{\cdot}t\ a'{\cdot}\mathtt{l}\ e) \to (\mathtt{if}\ a{\cdot}t\ a'{\cdot}\mathtt{l}\ |e|){\cdot}\mathtt{l}}\ [\text{THEN}]$$

$$\frac{\delta;\pi_{\mathbb{C}}(\Theta);\Gamma \vdash a \to e}{\delta;\Theta;\Gamma \vdash \mathbb{C}[a] \to \mathbb{C}[e]}\ [\text{CONG}] \qquad \frac{}{\delta;\Theta;\Gamma \vdash (\mathtt{if}\ a{\cdot}f\ e\ a'{\cdot}\mathtt{l}) \to (\mathtt{if}\ a{\cdot}f\ |e|\ a'{\cdot}\mathtt{l}){\cdot}\mathtt{l}}\ [\text{ELSE}]$$

$$\frac{}{\delta;\Theta;\Gamma \vdash \mathtt{l} \to \mathtt{l}{\cdot}\mathtt{l}}\ [\text{LOCAL}] \qquad \frac{\pi_{(\mathtt{nbr}\ [])}(\Theta) = (\overline{\delta} \mapsto \overline{a}{\cdot}\overline{\mathtt{l}}) \quad \phi = (\overline{\delta} \mapsto \overline{\mathtt{l}}, \varepsilon(\mathtt{uid}) \mapsto \mathtt{l})}{\delta;\Theta;\Gamma \vdash (\mathtt{nbr}\ a{\cdot}\mathtt{l}) \to (\mathtt{nbr}\ a{\cdot}\mathtt{l}){\cdot}\phi}\ [\text{NBR}]$$

$$\frac{}{\delta;\Theta;\Gamma \vdash \mathtt{x} \to \mathtt{x}{\cdot}\Gamma(\mathtt{x})|_{\mathbf{dom}(\Theta),\varepsilon(\mathtt{uid})}}\ [\text{VAR}] \qquad \frac{\delta;\pi_{(\mathtt{rep}^{\mathring{\mathtt{l}}_1}\ \mathtt{x}\ \mathtt{w}\ [])}(\Theta);\Gamma, (\mathtt{x} := (\Gamma(\mathtt{w}) \rhd \mathring{\mathtt{l}}_1)) \vdash a \to a'{\cdot}\mathring{\mathtt{l}}_2}{\delta;\Theta;\Gamma \vdash (\mathtt{rep}^{\mathring{\mathtt{l}}_1}\ \mathtt{x}\ \mathtt{w}\ a) \to (\mathtt{rep}^{\mathring{\mathtt{l}}_1 \rhd \mathring{\mathtt{l}}_2}\ \mathtt{x}\ \mathtt{w}\ a'{\cdot}\mathring{\mathtt{l}}_2){\cdot}\mathring{\mathtt{l}}_2}\ [\text{REP}]$$

$$\frac{}{\delta;\Theta;\Gamma \vdash (\mathtt{o}\ \overline{a}{\cdot}\overline{v}) \to (\mathtt{o}\ \overline{a}{\cdot}\overline{v}){\cdot}\varepsilon(\mathtt{o}, \overline{v})}\ [\text{OP}] \qquad \frac{\delta;\pi_{(\mathtt{d}^{[]}\ \overline{a}{\cdot}\overline{v})}(\Theta);(args(\mathtt{d}) := \overline{v}) \vdash (body(\mathtt{d}) \rhd \mathtt{s}) \to a{\cdot}\mathring{v}}{\delta;\Theta;\Gamma \vdash (\mathtt{d}^{\mathtt{s}}\ \overline{a}{\cdot}\overline{v}) \to (\mathtt{d}^{a}\ \overline{a}{\cdot}\overline{v}){\cdot}\mathring{v}}\ [\text{FUN}]$$

Figure 5: CFC operational semantics

the domain of a field ϕ to a given set of devices $\overline{\delta}$, which we denote as $\phi|_{\overline{\delta}}$. When restriction is applied to local values it works as the identity function. A tree environment, Θ, maps devices to runtime expressions (namely, it keeps neighbour trees); a variable environment, Γ, maps variables to runtime values. Since tree environments and variable environments are also mappings, all the above conventions and operators will be used for them as well.

5.2. Congruence contexts and alignment contexts

The operational semantics uses *congruence contexts*, ranged over by $\mathbb{C}$, to impose an order of evaluation of subexpressions in an orthogonal way with respect to the actual semantic rules; and it uses *alignment contexts*, ranged over by $\mathbb{A}$, to properly navigate into evaluation trees. In particular, note that $\mathbb{C}$ is a subcase of $\mathbb{A}$ (see Figure 5).

A context $\mathbb{A}$ is an auxiliary runtime expression with a *hole* []. As usual, we write $\mathbb{A}[e]$ to denote the runtime expression obtained by filling the hole of $\mathbb{A}$ with the runtime expression e. If a given runtime expression e matches $\mathbb{C}[e']$, then e' is the next subexpression of e where evaluation will occur, positioned in e as [] is positioned in $\mathbb{C}$. The way the syntax of congruence contexts $\mathbb{C}$ is structured constrains the operational semantics to evaluate the first argument of `if` and then, depending on its outcome, the second or third, and to non-deterministically evaluate arguments in function and operation calls. For instance, the runtime expression `(* 1·1 (+ 2·2 3))` matches $\mathbb{C}'[e']$ only by $\mathbb{C}' =$ `(* 1·1 [])` and $e' =$ `(+ 2·2 3)`: this means that e' contains the next subexpression to evaluate. The expression e', in turn, matches $\mathbb{C}''[e'']$ only by $\mathbb{C}'' =$ `(+ 2·2 [])` and $e'' =$ `3`. Therefore 3 is the next subexpression to evaluate (becoming $3 \cdot 3$).

5.3. Auxiliary functions

The projection operator π implements the mechanism for synchronising navigation of an evaluation tree with those of neighbour trees. Namely, $\pi_{\mathbb{A}}(\Theta)$ takes a tree environment Θ and extracts a new tree environment obtained by discarding the trees that do not match the alignment context $\mathbb{A}$ (according to the *alignment context matching relation* "::") and extracting the corresponding subtree matching the hole in the remaining ones. As an example, given $\Theta_0 = (\ \delta_1 \mapsto (\texttt{if}\ a{\cdot}t\ e_1\ e_2){\cdot}v_1,\ \delta_2 \mapsto (\texttt{if}\ a'{\cdot}f\ e_3\ e_4){\cdot}v_2\ )$ and $\mathbb{A} = (\texttt{if}\ a'{\cdot}t\ []\ e_2')$, we have $\pi_{\mathbb{A}}(\Theta_0) = (\delta_1 \mapsto e_1)$. In fact, the evaluation tree for δ_2 is discarded since it does not match $\mathbb{A}$ due to the label of first argument being f, while the evaluation tree for δ_1 matches and extracts e_1.

The replacement operator $\triangleright$ retains the right-hand side if this is not empty, otherwise it takes the left-hand side. It is useful to handily update null decorations.

To take into account special constants, mathematical operations, usual abstract data types operations, and context-dependent operators, we introduce a special auxiliary function ε, whose actual definition is abstracted away. This is such that $\varepsilon(\texttt{o}, \overline{v})$ computes the result of applying built-in operator `o` to values $\overline{v}$ in the current environment of the device. In particular, we assume constant `self` gets evaluated to the current device identifier. The ε function also encapsulates measurement variables such as `nbr-range` and `dt` and interactions with the external world via sensors and actuators. In order not to escape the domain restricted by operator `if`, as discussed in Sections 2 and 3, for each built-in operator `o` we assume that:

(i) $\varepsilon(\texttt{o}, v_1, \cdots, v_n)$ is defined (i.e., its evaluation does not get stuck) only if all the field values in $v_1, \ldots, v_n$ have the same domain; and

(ii) if $\varepsilon(\texttt{o}, v_1, \cdots, v_n)$ returns a field value ϕ and there is at least one field value v_i in $v_1, \ldots, v_n$, then $\mathbf{dom}(\phi) = \mathbf{dom}(v_i)$.

5.4. Reduction rules

Following [26], we formulate the reduction relation by means of reduction rules (which may be applied at any point in an expression) and congruence rules (which express the fact that if $e \to e'$ then $(\texttt{o}\ e_1 \ldots e_{i-1}\ e\ e_{i+1} \ldots e_n) \to (\texttt{o}\ e_1 \ldots e_{i-1}\ e'\ e_{i+1} \ldots e_n)$, and so on). The reduction relation is of the form $\boxed{\delta; \Theta; \Gamma \vdash e \to e'}$, to be read "expression e reduces to expression e' in one step", where δ is the local device, Θ is the current tree environment and Γ is the current store of variables (which is built incrementally in each reduction step by the congruence rules [REP] and [FUN] when evaluation enters the third argument of a `rep`-expressions or the body of a function, respectively).

The reduction relation models the execution of a single computation round, computed as $\delta; \Theta; \bullet \vdash a \to^* a'{\cdot}v$ where: δ is the local device; Θ is the set of evaluation trees produced by neighbours at their prior computation round; the variable environment is empty (the main expression must not contain free variables); and a is the runtime expression resulting from the computation of previous round with all the annotations (not superscripts) erased—at the very first round a is simply the top-level surface expression. During computation steps the run-time expression will be decorated with annotations, until one appears at the top level in the final runtime expression $a'{\cdot}v$, where v represents the local value of the computational field currently being computed. Some superscripts will be present at the end of the round, for they represent the side-effect of computation on the evaluation tree that should be transferred to next round. In particular, as already mentioned:

(i) the final runtime expression $a'{\cdot}v$ will be shipped to neighbours, replacing there the one previously sent;

(ii) the runtime expression obtained from $a'\cdot v$ by dropping all annotations (not superscripts), denoted by $init(a'\cdot v)$, will be used as starting point for the next round computation.

We now describe each reduction rule in turn. Computation rules have a common pattern: they compute a result value v, which appears as top-level annotation—in the following we shall say that v is the "local result". Rule [LOCAL] simply identically annotates a local value. Rule [VAR] looks at the value $\Gamma(\mathtt{x})$ associated to $\mathtt{x}$ by the variable environment, and (in case it is a field) restricts it to the set of currently aligned neighbours $\overline{\delta}$ (plus the local device $\varepsilon(\mathtt{uid}) = \delta$). Rule [NBR] is the one actually gathering values from Θ: let $\mathtt{l}$ be the value locally computed, we extract the corresponding values $\overline{\mathtt{l}}$ from aligned neighbours $\overline{\delta}$, and use as local result the corresponding field $\overline{\delta} \mapsto \overline{\mathtt{l}}$ (adding the local slot $\delta \mapsto \mathtt{l}$). Rule [OP] computes the result of applying operator $\mathtt{o}$ to values $\overline{v}$ (done by function ε, which gives semantics to operators), to be used as a local result. Rules [THEN] and [ELSE] handle condition branching: rule [THEN] (resp. [ELSE]) uses the label of second (resp. third) argument as local result in case of positive (resp. negative) condition, and erases the other branch (which may contain superscripts generated in the previous round).

Rule [CONG] can be understood as a compact representation for six different congruence rules, corresponding to the 6 cases for the context $\mathbb{C}$. While navigating the evaluation tree inside context $\mathbb{C}$ to identify the next evaluation site a (which should be non-annotated), this rule contemporarily enters the same context into all slots of the tree environment Θ, guaranteeing that the expression to evaluate is kept synchronised with the corresponding trees in Θ. Note that rule [CONG] does not describe the congruence rules for `rep`-expressions and function applications. In fact, the metavariable $\mathbb{C}$ does not range over contexts of the form $(\mathtt{rep}^{\mathring{1}}\ \mathtt{x}\ \mathtt{w}\ [])$ and $(\mathtt{d}^{[]}\ \overline{a}\cdot\overline{v})$. The rationale for this choice is that the corresponding rules, [REP] and [FUN], need to update the variable environment Γ by adding to Γ the `rep`-bound variable $\mathtt{x}$ or by completely replacing Γ with the environment for the function formal parameters $args(\mathtt{d})$, respectively. Moreover, [REP] and [FUN] are not pure congruence rules: each of them encodes a congruence rule possibly followed by a computation rule.[4] Note that this encoding exploits the notation $\mathring{y}$ and the auxiliary function $\triangleright$ defined above.

Rule [REP] handles evolution of a field. When the superscript $\mathring{\mathtt{l}}_1$ is null, the evaluation of the body of a `rep`-expression is carried on in an environment that assigns to the `rep`-bound variable $\mathtt{x}$ the value of the variable or local value $\mathtt{w}$—with abuse of notation we indicate it as $\Gamma(\mathtt{w})$: when $\mathtt{w}$ is a local value $\mathtt{l}$ we assume $\Gamma(\mathtt{l}) = \mathtt{l}$. When the superscript $\mathring{\mathtt{l}}_1$ is a local value $\mathtt{l}_1$, the evaluation of the body of `rep`-expression is carried on in an environment that assigns to the `rep`-bound variable $\mathtt{x}$ the value $\mathtt{l}_1$. If the reduction step performed (in the premise of the rule) on the body of the `rep`-expression produces an evaluated runtime expression (i.e., if the annotation $\mathring{\mathtt{l}}_2$ is not null), then the local result is propagated to the `rep`-expression (which becomes evaluated).

When the actual parameters of a function call are evaluated, rule [FUN] performs a reduction step on the function body in an environment consisting of the proper association of formal parameters $args(\mathtt{d})$ to values $\overline{v}$: the (possibly null) resulting annotation $\mathring{v}$ is transferred as local result. If the superscript $\mathtt{s}$ is null, replacement operator $\triangleright$ guarantees the function body is used instead.

6. Well-formedness

This section introduces the problem of well-formed specifications, and the need for a sound type system as developed in Section 7. A field expression is considered ill-formed if its evaluation (according to the operational semantics) can at some point get stuck: a situation that in an actual execution would result in a run-time error. Correspondingly, a function is ill-formed if its application to some argument forms a ill-formed expression, and a whole program is ill-formed if its functions or main expression are ill-formed. The following examples clarify the key notion of ill-formed functions.

Example 6.1. Function

```
(def wrong-channel-avoiding-obstacles (x src dst width)    ;; can not be typed
  (channel-avoiding-obstacles (nbr x) src dst width))
```

using the function `channel-avoiding-obstacles`, is ill-formed. This is due to the fact that the field value ϕ, which is produced by `(nbr x)` and passed into `channel-avoiding-obstacles`, conflicts with its use as the first input

[4] When the annotation $\mathring{\mathtt{l}}_2$ in rule [REP] is not null, and when the annotation $\mathring{v}$ in rule [FUN] is not null.

to **if**, which requires a Boolean value. Rules [THEN] and [ELSE] thus cannot be applied, and the evaluation cannot be completed.

Example 6.2. The function

```
(def wrong-f-two (x)    ;; can not be typed
  (min-hood (min-hood (nbr (nbr x)))))
```

which tries to find the minimum value of `x` within two hops, is ill-formed. When the function is applied to a local value, the body fails to evaluate because Rule [NBR] requires its input to be a local value, and thus cannot be applied to the outer **nbr**. This prevents the need to communicate a field value whose size scales linearly with the number of neighbours, which might be extremely burdensome. A well-formed alternative that produces the same computational result as `wrong-f-two` is intended to is

```
(def right-f-two (x)    ;; has type: (β) → β
  (min-hood (nbr (min-hood (nbr x)))))
```

This takes advantage of the commutative property of minimisation to break the minimisation into two stages, thus avoiding the communication explosion of the ill-formed formulation.

Example 6.3. The function

```
(def wrong-nbr-if (x y z)    ;; can not be typed
  (min-hood (-[f,f] (if (sense 1) (nbr x) (nbr y)) (nbr z))))
```

is ill-formed. Its body will fail to evaluate on Rules [THEN] and [ELSE], since they require local values for the test and returned values. This prevents conflicts between field domains, as in this case, where the field produced by `(nbr z)` would contain all neighbours, while the field produced by the **if** expression would contain only a subset, leaving the fields mismatched in domain at the subtraction. A correct alternative is

```
(def right-nbr-if (x y z)    ;; has type: (num, num, num) → num
  (min-hood (-[f,f] (nbr (if (sense 1) x y)) (nbr z))))
```

which conducts the test locally, ensuring that the domains of the two fields match.

7. Typing and properties

This section presents a type inference system, used to intercept ill-formed programs, as those shown in Section 6. Technically, this type system is designed to guarantee the following two properties:

Domain Alignment The domain of every field value arising during the reduction of a well-typed expression consists of the identifiers of the aligned neighbours and of the identifier of the `uid` device.

Type Soundness The reduction of a well-typed expression does not get stuck.

Note that domain alignment guarantees a proper handling of restriction which, in turn, is needed for type soundness. Note as well that the type system does not guarantee that evaluation will complete: for example, an expression such as `(def nohalt (x) (nohalt (+ x 1)))`, which infinitely recurses, is well-typed with type $(\texttt{num}) \to \texttt{num}$.

7.1. Typing rules for surface programs

The type system builds on the Hindley-Milner type system [17] for ML-like functional languages. Since the type inference rules are syntax-directed, they straightforwardly specify a variant of the Hindley-Milner type inference algorithm [17]. I.e., they specify an algorithm (not present here) that, given an expression `e` and type assumptions for its free variables, either fails (if the expression cannon be typed under the given type assumptions) or returns its *principal type*, i.e., a type such that all the types that can be assigned to `e` by the type inference rules can be obtained from the principal type by substituting type variables with types.

Types are partitioned in two kinds: *expression types* (for expressions) and *function types* (for built-in operators and user-defined functions). The former are further partitioned in two kinds: *local types* (for fields of local values, e.g., that produced by `if`) and *field types* (for fields of field values, e.g., that produced by `nbr`). Moreover, in order to support polymorphic use of built-in operators and user-defined functions, we consider also *function type schemes*.

The syntax of expression types, local types, field types, function types and function type schemes is given in Fig. 6 (top). An expression type E is either a *type variable* α, or a local type, or a field type. A local type L is either a *local type variable* β, or the type of Booleans `bool`, or the type of numbers `num`, or the type of a pair value $\langle L, L\rangle$. A field type Φ is the type $\texttt{field}(L)$ of a field whose domain contains values of local type L.

A function type F is the type $(\overline{E}) \to E$ of a (possibly zero-arity) built-in operator or user-defined function. We write $\mathbf{FTV}(F)$ to denote free type variables and the free local variables occurring in F. E.g., $\mathbf{FTV}((\alpha, \texttt{field}(\beta)) \to \langle \alpha, \beta\rangle) = \alpha, \beta$. Function type schemes, ranged over by FS, support typing polymorphic uses of built-in operators and user defined-functions. Namely, each built-in operator or user-defined function `f` has a *function type scheme* $\forall\overline{\alpha}\overline{\beta}.F$, where $\overline{\alpha}$ and $\overline{\beta}$ are all the type variables and local type variables occurring in the type F, respectively. Each use of `f` can be typed with any type obtained from $\forall\overline{\alpha}\overline{\beta}.F$ by replacing the type variables $\overline{\alpha}$ with types and the local type variables $\overline{\beta}$ with local types.

Expression type environments, ranged over by $\mathscr{X}$ and written $\overline{\texttt{x}} : \overline{E}$, are used to collect type assumptions for program variables (i.e., the formal parameters of the functions and the variables introduced by the `rep`-construct). *Type-scheme environments*, ranged over by $\mathscr{D}$ and written $\overline{\texttt{d}} : \overline{FS}$, are used to collect the function-type schemes inferred for the user-defined functions. The distinguished *built-in type-scheme environment* $\mathscr{O}$ associates a function-type scheme to each built-in function `o`—Fig. 7 shows the local type schemes for the built-in functions used in the examples presented through the paper.

The type infeence rules are given in Fig. 6 (bottom). The typing judgement for expressions is of the form "$\mathscr{D}; \mathscr{X} \vdash \texttt{e} : E$", to be read: "`e` has type E under the (implicit) type scheme assumption $\mathscr{O}$ (for built-in operators), the type-scheme assumptions $\mathscr{D}$ (for user-defined functions), and the expression type assumptions $\mathscr{X}$ (for the program variables occurring in `e`), respectively". As a standard syntax in type systems [26], given $\overline{E} = E_1, \ldots, E_n$ and $\overline{\texttt{e}} = \texttt{e}_1, \ldots, \texttt{e}_n$ ($n \geq 0$), we write $\mathscr{D}; \mathscr{X} \vdash \overline{\texttt{e}} : \overline{E}$ as short for $\mathscr{D}; \mathscr{X} \vdash \texttt{e}_1 : E_1 \cdots \mathscr{D}; \mathscr{X} \vdash \texttt{e}_n : E_n$.

Rule [ST-VAR] (for variables) lookups the type assumptions for `x` in $\mathscr{X}$.

Rule [ST-LOCAL] (for local values) exploits the auxiliary function *typeof*, defined in Fig. 6 (middle).

Rule [ST-OP] (for built-in function application) and Rule [ST-FUN] (for user-defined function application) are standard. They ensures that the function-type scheme $\forall\overline{\alpha}\overline{\beta}.F$ associated to the built-in function or user-defined function name being applied is instantiated by substituting the local type variables $\overline{\beta}$ with local types $\overline{L}$ and the type variables $\overline{\alpha}$ with (possibly non-local) types $\overline{E}$.

Rule [ST-REP] (for `rep`-expressions) ensures that both the variable `x`, its initial value `w` and the body `e` have (the same) local type. This prevents a device δ to store in `x` a field value ϕ (whose domain is by construction equal to the subset of the neighbours of δ that are aligned—i.e., that have evaluated this `rep`-expession in their last evaluation round) that may be reused in the next computation round of δ, when the set of aligned neighbours may be different from the domain of ϕ (this would cause a domain alignment error).

Rule [ST-NBR] (for `nbr`-expressions) ensures that the body `e` of the expression has a local type. This prevents the attempt to create a "field of fields" (i.e., a field that maps device identifies to fields values)—that would be ill-formed.

Rule [ST-IF] (for `if`-expressions) ensures that the condition $\texttt{e}_0$ has type `bool` and the branches $\texttt{e}_1$ and $\texttt{e}_2$ have (the same) local type. This prevents the `if`-expression to evaluate to a field value whose domain is different from the subset of the neighbours of δ that are aligned—i.e., that have evaluated this `if`-expression in their last evaluation round. For instance, the expression

Expression types:

$E ::= \alpha \mid L \mid \Phi$ — expression type

$L ::= \beta \mid \texttt{bool} \mid \texttt{num} \mid \langle L, L \rangle$ — local type

$\Phi ::= \texttt{field}(L)$ — field type

Function types and function type schemes:

$F ::= (\overline{E}) \to E$ — function type

$FS ::= \forall \overline{\alpha}\overline{\beta}.F$ — function type scheme

Auxiliary function for typing values:

$typeof(\texttt{b}) = \texttt{bool}$

$typeof(\texttt{n}) = \texttt{num}$

$typeof(\langle \texttt{l}_1, \texttt{l}_2 \rangle) = \langle typeof(\texttt{l}_1), typeof(\texttt{l}_2) \rangle$

Surface expression typing: $\boxed{\mathscr{D}; \mathscr{X} \vdash \texttt{e} : E}$

$$\frac{}{\mathscr{D}; \mathscr{X}, \texttt{x} : E \vdash \texttt{x} : E}\ [\text{ST-VAR}] \qquad \frac{L = typeof(\texttt{l})}{\mathscr{D}; \mathscr{X} \vdash \texttt{l} : L}\ [\text{ST-LOCAL}]$$

$$\frac{\mathscr{O}(\texttt{o}) = \forall \overline{\alpha}\overline{\beta}.F \qquad (\overline{E}) \to E = F[\overline{\alpha} := \overline{E}][\overline{\beta} := \overline{L}] \qquad \mathscr{D}; \mathscr{X} \vdash \overline{\texttt{e}} : \overline{E}}{\mathscr{D}; \mathscr{X} \vdash (\texttt{o}\ \overline{\texttt{e}}) : E}\ [\text{ST-OP}]$$

$$\frac{\mathscr{D}(\texttt{d}) = \forall \overline{\alpha}\overline{\beta}.F \qquad (\overline{E}) \to E = F[\overline{\alpha} := \overline{E}][\overline{\beta} := \overline{L}] \qquad \mathscr{D}; \mathscr{X} \vdash \overline{\texttt{e}} : \overline{E}}{\mathscr{D}; \mathscr{X} \vdash (\texttt{d}\ \overline{\texttt{e}}) : E}\ [\text{ST-FUN}]$$

$$\frac{\mathscr{D}; \mathscr{X} \vdash \texttt{w} : L \qquad \mathscr{D}; \mathscr{X}, \texttt{x} : L \vdash \texttt{e} : L}{\mathscr{D}; \mathscr{X} \vdash (\texttt{rep x w e}) : L}\ [\text{ST-REP}] \qquad \frac{\mathscr{D}; \mathscr{X} \vdash \texttt{e} : L}{\mathscr{D}; \mathscr{X} \vdash (\texttt{nbr e}) : \texttt{field}(L)}\ [\text{ST-NBR}]$$

$$\frac{\mathscr{D}; \mathscr{X} \vdash \texttt{e}_0 : \texttt{bool} \qquad \mathscr{D}; \mathscr{X} \vdash \texttt{e}_1 : L \qquad \mathscr{D}; \mathscr{X} \vdash \texttt{e}_2 : L}{\mathscr{D}; \mathscr{X} \vdash (\texttt{if e}_0\ e_1\ \texttt{e}_2) : L}\ [\text{ST-IF}]$$

Function typing: $\boxed{\mathscr{D} \vdash \texttt{D} : FS}$

$$\frac{\mathscr{D}, \texttt{d} : \forall \bullet.\overline{E} \to E; \overline{\texttt{x}} : \overline{E} \vdash \texttt{e} : E \qquad \overline{\alpha}\overline{\beta} = \mathbf{FTV}((\overline{E}) \to E)}{\mathscr{D} \vdash (\texttt{def d}(\overline{\texttt{x}})\ \texttt{e}) : \forall \overline{\alpha}\overline{\beta}.(\overline{E}) \to E}\ [\text{ST-FUNCTION}]$$

Program typing: $\boxed{\vdash \texttt{P} : L}$

$$\frac{\begin{array}{l}\mathscr{D}_0 = \bullet \\ \texttt{D}_i = (\texttt{def d}_i(\_)\ \_) \qquad \mathscr{D}_{i-1} \vdash \texttt{D}_i : FS_i \qquad \mathscr{D}_i = \mathscr{D}_{i-1}, \texttt{d}_i : FS_i \qquad (i \in 1..n) \\ \mathscr{D}_n; \emptyset \vdash \texttt{e} : L\end{array}}{\vdash \texttt{D}_1 \cdots \texttt{D}_n\ \texttt{e} : L}\ [\text{ST-PROGRAM}]$$

Figure 6: CFC: expression types, function types and function schemes; auxiliary function for typing values; and type inference rules for surface expressions, function declarations, and programs

```
(min-hood (+[f,f]  (if (o1) (nbr (o2)) (nbr (o3)))  (nbr (o4))))
```

Pure built-in function (independent from the current device and value-tree environment):

$$
\begin{array}{lcl}
\mathscr{O}(\texttt{and[f,f]}) & = & (\texttt{field(bool)},\texttt{field(bool)}) \to \texttt{field(bool)} \\
\mathscr{O}(\texttt{any-hood}) & = & (\texttt{field(bool)}) \to \texttt{bool} \\
\mathscr{O}(\texttt{min-hood}) & = & \forall\beta.(\texttt{field}(\beta)) \to \beta \\
\mathscr{O}(\texttt{mux}) & = & \forall\beta.(\texttt{bool},\beta,\beta) \to \beta \\
\mathscr{O}(\texttt{pair}) & = & \forall\beta_1\beta_2.(\beta_1,\beta_2) \to \langle\beta_1,\beta_2\rangle \\
\mathscr{O}(\texttt{snd}) & = & \forall\beta_1\beta_2.(\langle\beta_1,\beta_2\rangle) \to \beta_2 \\
\mathscr{O}(\texttt{sense}) & = & (\texttt{num}) \to \texttt{bool} \\
\mathscr{O}(\texttt{sum-hood}) & = & (\texttt{field(num)}) \to \texttt{num} \\
\mathscr{O}(\texttt{+}) & = & (\texttt{num},\texttt{num}) \to \texttt{num} \\
\mathscr{O}(\texttt{+[f,l]}) & = & (\texttt{field(num)},\texttt{num}) \to \texttt{field(num)} \\
\mathscr{O}(\texttt{-}) & = & (\texttt{num},\texttt{num}) \to \texttt{num} \\
\mathscr{O}(\texttt{-[f,f]}) & = & (\texttt{field(num)},\texttt{field(num)}) \to \texttt{field(num)} \\
\mathscr{O}(\texttt{*}) & = & (\texttt{num},\texttt{num}) \to \texttt{num} \\
\mathscr{O}(\texttt{/}) & = & (\texttt{num},\texttt{num}) \to \texttt{num} \\
\mathscr{O}(\texttt{=[l,f]}) & = & \forall\beta.(\beta,\texttt{field}(\beta)) \to \texttt{field}(\beta)
\end{array}
$$

Non-pure built-in function (depend from the current device and value-tree environment):

$$
\begin{array}{lcl}
\mathscr{O}(\texttt{dt}) & = & () \to \texttt{num} \\
\mathscr{O}(\texttt{nbr-range}) & = & () \to \texttt{field(num)} \\
\mathscr{O}(\texttt{uid}) & = & () \to \texttt{num}
\end{array}
$$

Figure 7: CFC: function type schemes for the built-in functions used in the examples through the paper

(where `o1` has type `() → bool`, and `o2`, `o3`, `o4` have type `() → num`) is ill-typed. Its body will fail to type-check because of type rule [ST-IF], and would cause a run time error, since evaluation rules [THEN] and [ELSE] require local values to be produced by the evaluation of the branches of an `if`-expression. This prevents conflicts between field domains, as in this case, where the field produced by `(nbr (o4))` would contain all neighbours, while the field produced by the `if`-expression would contain only a subset, leaving the fields mismatched in domain at the sum. A correct alternative is

```
(min-hood (+[f,f] (nbr (if (o1) (o2) (o3))) (nbr (o4))))
```

which conducts the test locally, ensuring that the domains of the two fields match.

Function declaration typing (represented by judgement "$\mathscr{D} \vdash \texttt{D} : FS$") and program typing (represented by judgement "$\vdash \texttt{P} : L$") are almost standard.[5] We say that a program P is *well-typed* to mean that $\vdash \texttt{P} : L$ holds for some local type L.

Example 7.1. The CFC calculus captures all the examples presented through the paper. In particular, the type system it able to type the examples of well-formed programs illustrated in Sections 2, 4 and 6, and it rejects the examples of ill-formed programs illustrated in Section 6.

7.2. Domain alignment and type soundness

The state of the computation for a program $\texttt{P} = \overline{\texttt{D}}\,\texttt{e}$ in a device δ is represented by a *configuration*, which is a 3-tuple $\langle \delta, \Theta, e \rangle$ where $\delta \notin \mathbf{dom}(\Theta)$, $|e| = \texttt{e}$ and, for all $\delta' \in \mathbf{dom}(\Theta)$, $|\Theta(\delta')| = \texttt{e}$.

A configuration $\langle \delta, \Theta, e \rangle$

- is in *normal form* if $\delta; \Theta; \bullet \vdash e \not\rightarrow$; and

[5] To simplify he presentation, we have not considered the issue of typing mutually recursive user-defined functions. To extend the type system in order to type mutually recursive user-defined function can be straightforwardly accomplished by exploiting standard techniques (see, e.g, [32]).

- is *final* if $e = a{\cdot}v$.

Note that if a configuration is final then it is in normal form. A final configuration represents the state of a device when a round of computation has been successfully completed. A configuration that is in normal form and is not final represents a computation that is *stuck*, i.e., it represents a runtime error.

The set of *reachable tree environments* and the set of *reachable configurations* for the program $\mathtt{P} = \overline{\mathtt{D}}\,\mathtt{e}$ in a device δ (denoted by $RTE(\delta,\mathtt{P})$ and $RC(\delta,\mathtt{P})$, respectively) are the set of tree environments and the set of configurations inductively defined by the following clauses, taking into account the iterative nature of computation rounds:

- $(\delta_1 \mapsto e_1, \ldots, \delta_n \mapsto e_n) \in RTE(\delta,\mathtt{P})$ $(n \geq 0)$ if $\delta \notin \{\delta_1,...,\delta_n\}$ and for all $i \in 1..n$ there exists $\langle \delta_i, \Theta_i, e_i \rangle \in RC(\delta_i,\mathtt{P})$ (for some Θ_i) such that $\langle \delta_i, \Theta_i, e_i \rangle$ is final.
- $\langle \delta, \Theta, e \rangle \in RC(\delta,\mathtt{P})$ if $\Theta \in RTE(\delta,\mathtt{P})$ and
  - either $\langle \delta, \Theta, \mathtt{e} \rangle$ is a configuration and $\delta;\Theta;\bullet \vdash \mathtt{e} \rightarrow^* e$;
  - or there exists $\langle \delta, \Theta', e' \rangle \in RC(\delta,\mathtt{P})$ (for some Θ') such that $\langle \delta, \Theta', e' \rangle$ is final and $\delta;\Theta;\bullet \vdash \mathit{init}(e') \rightarrow^* e$.

Reachable configurations like $\langle \delta, \Theta, \mathtt{e} \rangle$ and $\langle \delta, \Theta, \mathit{init}(e') \rangle$ are called *initial* configurations. In particular, configurations like $\langle \delta, \Theta, \mathtt{e} \rangle$ are called *bootstrap* configurations. An initial configuration represents the state of a device that is ready for starting a round of computation, and a bootstrap configuration may represent the state of a device that is ready for starting its first round of computation (note that, if the main expression `e` does not contain `rep`-expressions, `if`-expression and function calls, then any initial configuration is a bootstrap configuration). Therefore, reachable configurations are the configurations that a device can assume when it start from a bootstrap configuration and interacts only with devices that have started from a bootstrap configuration. I.e., reachable configurations are the possible states of the devices in a network initially empty and such that whenever a new device appears it has a bootstrap configuration.

We are now able to formally state the domain alignment and type soundness properties (cf. the explanation at the beginning of Section 7).

Theorem 7.2 (Domain Alignment). *For every well-typed program* `P` *and every device* δ*, if* $\langle \delta, \Theta, e \rangle \in RC(\delta, P)$ *and* $\delta;\Theta;\bullet \vdash e \rightarrow e'$ *(for some* e'*) then every subreduction* $\delta;\Theta_0;\Gamma \vdash e_1 \rightarrow e_2$ *in the reduction* $\delta;\Theta;\bullet \vdash e \rightarrow e'$ *is such that if* $e_2 = a{\cdot}\phi$ *then* $\boldsymbol{dom}(\phi) = \boldsymbol{dom}(\Theta_0) \cup \{\delta\}$.

Theorem 7.3 (Type Soundness). *For every well-typed program* `P` *and every device* δ*, if* $\langle \delta, \Theta, e \rangle \in RC(\delta, P)$ *and* $\delta;\Theta;\bullet \vdash e \not\rightarrow$*, then* $\langle \delta, \Theta, e \rangle$ *is final.*

The proofs of Theorems 7.2 and 7.3 are given in Appendix A. The proofs are done by introducing a suitable notion of typing for runtime expressions and configurations and showing that:

(i) the initial configuration of a well-typed program at the first round of computation is well typed (Lemmas A.2 and A.4);

(ii) the initial configuration of a well-typed program in all the subsequent rounds of computation is well typed (Lemmas A.3 and A.4);

(iii) the domain of every field value occurring in a well-typed configuration consists of the identifiers of the aligned neighbours and of the identifier of the `uid` device (Lemma A.5);

(iv) reducing a well-typed configuration produces a well-typed configuration (Theorem A.6 - subject reduction); and

(v) if a non-evaluated configuration is well typed then some reduction rule is always applicable (Lemma A.7 and Theorem A.8 - progress).

8. Toolchain

The computational field calculus presented in this paper is the foundation of a toolchain under development for spatial computers, encompassing *(i)* the Protelis Java-like language, *(ii)* various incarnations of the Protelis architecture, including in the Alchemist simulator and as a management toolkit for distributed servers, and *(iii)* self-stabilising "building block" libraries and higher-level programming APIs. This section presents a brief survey of these components (summarising relevant results from [39, 45, 40]), with the goal of illustrating how the theory of computational field calculus translates into practical impact for distributed applications.

8.1. Protelis Language

We have designed the Protelis language [39] as an implementation of the field calculus that is well-integrated with Java and the JVM framework. On the one hand, it is consistent with the specification of the field calculus, and its interpreter is based on the operational semantics described in this paper. On the other hand, it turns the field calculus into a fully blown programming language, by providing access to the rich Java API (semantically, a collection of built-in operators) as well as supporting mechanisms for first-class functions. In implementing Protelis, we have used the Xtext language development framework [22], which also supports a language-specific Eclipse IDE plugin that eases adoption and code writing by means of code highlighting, completion suggestions and compile-time error detection. Our complete Protelis implementation is free and open source, available at `http://protelis.github.io/`.

Syntactically, Protelis adopts a set of modern Java conventions, including using functional notation $f(e_1,\ldots,e_n)$ for function calls, infix notation for binary arithmetic and comparison operators, definition of local variables (using a **let** construct), anonymous functions of the form $(x_1,\ldots,x_n)->\{b\}$ applied to arguments by an **apply** construct, and import of static methods from Java APIs to be used as built-in operators. As an example, a Laplacian consensus function similar to that in Example 4.1, in which the `epsilon` value is selected randomly within a given range, could be implemented as a function `randomized-laplacian-consensus` as follows:

```
import java.lang.Math.random; ;; importing 'random' from Java

def consensus (init, f) {          ;; finding a consensus for init, using correction function f
  rep (val <- init) {
    val + f.apply(sumHood(nbr{val} - val))
  }
}

def randomized-laplacian-consensus (init, min, max) {
  let epsilon = min + (max - min) * random();    ;; computing epsilon
  consensus(init, (x) -> {epsilon * x})           ;; calling the general consensus function
}
```

8.2. Portable architecture

In addition to a language, the Protelis implementation also supplies a virtual machine architecture that links field calculus with the pragmatics of communication, execution, and interfacing with hardware, operating system, and other software [39]. This architecture is designed to make field calculus programs readily portable across simulation environments and also various classes of real networked devices (following the same general pattern as was used for the Proto VM [2]), since development and maintainability are greatly enhanced if the exact same code can used for execution at different stages of development, testing, and deployment.

Under this architecture, Protelis programs are first mapped into a valid representation of field calculus semantics. This is then executed by the Protelis interpreter at regular intervals, using abstract interfaces to communicate with other devices and to interact with the contextual environment (e.g., accessing sensors and position information, signalling actions by other programs). These abstract interfaces may then be instantiated for particular device platforms or simulations by setting when executions occur, how communication is implemented, and the methods used for accessing the environment. This architecture is implemented in Java, both to make it highly portable across JVM-based systems and devices (including the increasing number of low cost embedded devices that support Java), and also to exploit Java's reflection mechanisms in order to allow transparent use of a large variety of useful libraries and APIs.

To date, we have exercised this architecture through construction of two instantiations: one in the Alchemist framework [38] for simulation of large-scale spatially-embedded networks such as sensor networks, swarm robots, and pervasive computing scenarios; the other a daemon for coordinating management of networked services [16]. In the Alchemist instantiation, simulations are configured using a simple scripting language, which specifies a Protelis program as well as the collection of devices that will execute it, communication between those devices, and other aspects of the environment to be simulated. The Alchemist event-driven simulation engine then handles execution scheduling, message delivery, and updates to the environment tuple store. One of the key advantages in using Alchemist is its support for simulation of various realistic scenarios. For example, for indoor simulated environments, Alchemist can load images of building plans to be used as the simulated environment, and embeds various realistic movement models for people moving in those environments [25]. Complementarily, for outdoor simulated environments, it can load OpenStreetMap [24] data to be used as environment, GPS traces to place and move wearable devices associated with pedestrians, and GraphHopper[6] to support simulated movement of pedestrians. For instance, in [40] these features have been used to study how crowd steering algorithms could work in a simulation driven by GPS trace data collected at the 2013 Vienna City Marathon. The other instantiation applies Protelis to network service management. Here, each Protelis device lives on a server in an enterprise network, and is tethered to the networked service it is intended to manage by a service manager daemon. This daemon monitors the service, injecting information about its status and known dependencies into the environment and maintaining a neighborhood by opening parallel communication links to its corresponding daemons on any other servers that the monitored service communicates with. This can then be used for implementing service management applications, such as dependency-aware distributed recovery from faults [16].

8.3. Self-stabilising building blocks and APIs

Being functional in nature, both field calculus and its Protelis implementation promote a compositional and incremental approach to the construction of complex systems. As is typical software engineering practice, this can be approached by creating layers of APIs (libraries of function definitions) that progressively hide low-level details and provide programmers with powerful abstractions for structuring spatial computing systems, each uniformly seen as a reusable collective behaviour working on computational fields. Here, the key value of field calculus is its coherent semantics for the scoping and composition of distributed systems, which greatly simplifies the analysis of such APIs. This approach, as discussed in [45, 9], can support construction of distributed "building blocks" with formally proven resilience properties (which get transferred to the APIs and systems built on top) as well as creation of substitutable components (with the same function but trading off performance in different ways).

As a first layer, one can define combinators that capture "safe" patterns in the interplay of field calculus constructs, e.g., various ways in which **nbr** and **rep** combine to manage information across many hops in a network. Three useful such "building blocks" have been identified that can be used to completely hide **nbr** and **rep**, called `G`, `C` and `T` [9]. `G` is a "spreading" operation generalising broadcast: it executes the two tasks of computing shortest-path distances from a source and according to a given metric, and propagating values along the gradient of the distance using a suitable context-dependent accumulation function. Conversely, `C` is an "aggregation" operation: it accumulates information coming from an input field by progressively combining values that flow "downhill" on a potential field, using a suitable accumulation function. Finally, `T` is a "flexible timer" operator, making a pointwise value progressively decrease until reaching a "floor" value.

These three "building blocks," along with **if** and built-in operators, form a "sub-language" of self-stabilising systems, such that any system implemented using them is guaranteed to recover from any transitory change occurring in sensors and network topology [45]. On top of this, one can create various general-purpose or application-specific APIs, e.g., for collective spreading (`broadcast` to advertise events, `pathForecast` to forecast obstacles along a given direction, `distanceBetween` to compute and spread minimum distance between regions, `partition` to segregate space), for collective aggregation (`summarise` to sum all values of a field in a region, `average` to compute mean values), for time management (`limitedMemory` to hold a value until a deadline expires, `lowPassFilter` to smooth a rapidly changing field) and so on. Based on such APIs, as shown in [9], simple composition can allow an engineer to create complex applications, like e.g. for crowd management, in which spatial computing performed by smart devices

[6] `http://graphhopper.com/`

in large events can be used to detect crowded areas, and to provide directions for dispersal or for steering to desired destinations by circumventing dynamically forming crowds.

9. Related work

A large number of prior works present notions of computational fields; a thorough review may be found in [7]. Regarding the most similar: the Hood sensor network abstraction [54] and Butera's "paintable computing" hardware model [13] implement computational fields using only the local view, and thus do not ensure well-formed domains. Similar local views appear in a number of other approaches to computing with computational fields, such as TOTA [29], the chemical models in [47], and Meld [1]. A number of region-based approaches to sensor network programming provide an explicit model of fields, but either do not ensure well-formed domains (e.g., Abstract Regions [53]) or do provide well-formed domains (e.g., Regiment [34]) but not general computation. A number of parallel computing models also explicitly consider computational fields, most notably StarLisp [27] and systolic computing (e.g., [21, 41]). Both of these use a model of communication via parallel shifting of data on a structured network, which could be implemented by chained neighbourhood operations in field calculus. The $\sigma\tau$-Linda model [50] proposes an extension of Linda with a few constructs for spreading tuples to form fields, and adopting a notion of computation rounds very similar to the one we formalised. Similarly, the tiny calculus for fields in [43] has an analogous function style (which is in fact inspired by Proto again), but aims at defining behaviourally tractable fields, and hence it has much reduced expressiveness. More generally, while all of the key ingredients for programming computational fields are supported in a number of different languages (see [7]), at present only Proto supports all five that we found critical to include in the calculus. Critically, however, field calculus has been proven to be space-time universal [11], meaning that it is guaranteed to be able to express anything that can be expressed by these other computational field approaches. In some cases, this is straightforward (e.g., the $\sigma\tau$-Linda model can be implemented quite directly) while in others the relationship is more complex (e.g., the shift operators in StarLisp, which assume a perfectly crystalline spatial structure, coherent orientation relations, and tight time synchronization)

A number of other formal calculi have also been developed for parallel computations in structured environments, like 3π-calculus [14], Ambient calculus [15], and P-systems [37]: they all describe parallel computation over variously abstracted notions of space. As a key example, Milner's Bigraphs [31] have been proposed as a formal model to ground pervasive computing applications, focussing on handling a dynamic notion of topology and locality of interactions. Most specifically, works such as [12, 30] take bigraphs a step further easing the encoding of context-aware systems and deriving agent programs from BRS modelling. Although theoretically the Bigraphs model has sufficient expressiveness to capture any computational behaviour occurring on a dynamiclly evolving networked system, it does not capture the right abstractions to properly deal with computational fields—hence, expressing the examples we provided in this paper would be cumbersome. More generally, Bigraphs, and also the other calculi mentioned above, hardly express the functional nature of field computations, and do not provide means to reason in terms of aggregate-level descriptions of distributed behaviour. Correspondingly, they also lack the toolchain necessary to actually turn high-level system specifications into concrete systems.

A core operational semantics for discrete execution of Proto programs was developed in [44]. Although closely related to the present one, it was a preliminary attempt extremely limited in the types of computations it could represent, since it did not tackle the fully general problem of combining restriction, evolution, and recursive function calls (i.e. dynamically expanding evaluation trees), which we have addressed through the idea of aligning annotated evaluation trees. Based on [44], in [46] a full formalisation of discrete Proto was provided. This resulted in a rather large semantics aimed at a faithful representation of every construct in Proto and of their execution by the platform—e.g., including an intricate technique for optimising message size. The resulting model is then too complicated to readily use in proving language properties. In contrast, the operational semantics of CFC is general enough to cover all of Proto and many other spatial languages [7], and is compact enough to be a suitable basis for tackling interesting properties.

The syntax of CFC is a subset of the syntax of Proto (which, being based on the syntax of the programming language Lisp, is dynamically typed). We have defined a type inference system for CFC by building on the Hindley-Milner type inference system [17] for ML-like functional languages. The difference w.r.t. standard Hindley-Milner type inference is the distinction between local types and field types, which is crucial in order to guarantee domain alignment. Since we aimed to define a minimal core calculus we have considered a minimal set of primitive types

and type constructors (i.e., the types for booleans and numerical values, and pairs)—other primitive types and type constructors can be straightforwardly added. Note that the CFC calculus captures all the examples presented through the paper.

The type-soundness theorem (Theorem 7.3) is proved by using the standard technique of subject reduction and progress [55]. However, the nature the operational semantics (which models the computation performed by a single device at a single round of computation, which depends from the computation previously performed by the device's neighbours) introduced some challenging non-standard elements. For instance: the formalisation of the notion of well-formed tree environment (c.f. Section A.1) for dealing with the fact that each round of computation depends on the neighbour's evaluation trees; the formalisation of the domain alignment property (Theorem 7.2) which is also needed to prove type soundness (the proof of Theorem 7.3 uses the Lemma A.5); and the fact that the proof of the progress theorem (Theorem A.8) relies on the subject-reduction theorem (namely the proof of Lemma A.7 uses Theorem A.6)—this is due to fact that both the operational semantics rules [REP] and [FUN] encode a congruence rule possibly followed by a computation rule (cf. the discussion at the end of Section 5.4).

10. Conclusion and future work

This paper presents for the first time a minimal calculus with a sound type system for spatial computing models. We believe that our notion of static type system guaranteeing domain alignment and type soundness bootstraps investigations on other important properties.

- A first line of research concerns studying behavioural properties, identifying fragments of the calculus guaranteed to generate field computations with given properties. An initial work in this direction, as mentioned also in Section 8, has been developed in [45], which identifies a fragment of CFC enjoying self-stabilisation, and paving the way towards lightweight engineering of complex field computations. This work can be extended to provide a larger fragment guaranteeing self-stabilisation, as well as addressing new properties, e.g., addressing scale-independency of computations (programs that have a predictable conformation of aggregate-level behavior independent on the topology/density/timing of devices), and addressing dynamic aspects that pertain how quickly field computations repair to changes.
- A second line of research addresses the problem of universality and expressiveness. In [11] we proved CFC to be space-time universal, i.e., that it can approximate any physically realizable program over continuous or discrete space and time. This work can be extended in many ways to address important technical problems. First, there is a need for deeper understanding of the relationship between continuous specifications and discrete realizations. Second, it may be interesting to compare expressiveness of different fragments of the calculus, also against similar formalisms adopting different primitives. In [18] we introduced an higher-order extension of CFC. Higher-order supports code mobility and provides more expressiveness, however it opens new issues related to type-soundness and domain alignment. In future work we would like to prove type-soundness and domain alignment for higher-order CFC.
- A third line of investigation is more practical and addresses development of the toolchain discussed in Section 8. As the Protelis language and a basic supporting platform have been bootstrapped, next steps include adding a type inference system to Protelis along the lines of the contribution of the present paper, as well as extending to automatically checking self-stabilisation and other properties, improve platform support to address scalability and cloud-based techniques, and finally to build rich APIs for rapid and effective engineering of spatial computing systems.

In the long term, we believe CFC will serve as an important step toward identifying an engineering methodology for developing spatial computing and coordination systems able to make use of complex yet predictably well-behaved self-organising mechanisms [36], both in today's and in emergent distributed computing scenarios.

Acknowledgment

We thank the anonymous FOCLASA and SCP referees for many useful comments and suggestions for improving the presentation.

A. Proofs of Theorems 7.2 and 7.3

We say that: an expression type is *closed* to mean that it does not contain type variables and local type variables; and a typing judgement $\mathscr{D};\mathscr{X} \vdash \mathtt{e} : E$ is *closed* to mean that E is closed and all the expression types in the range of $\mathscr{X}$ are closed.

In the following we consider a well-typed program $\mathtt{P} = \overline{\mathtt{D}}\,\mathtt{e}_{\mathrm{main}}$. The main expression $\mathtt{e}_{\mathrm{main}}$ of $\mathtt{P}$ is closed expression and has a closed local type. Therefore, its typing derivation contains only closed typing judgements. In the following: we write $\mathscr{X} \vdash \mathtt{e} : E$ as short for the closed typing judgement $\mathscr{D};\mathscr{X} \vdash \mathtt{e} : E$, where the type-scheme environment $\mathscr{D}$ (which is univocally determined by $\overline{\mathtt{D}}$) is left implicit; and always consider closed expression types and closed expression-type environments. Moreover, given closed function type F', built in operator $\mathtt{o}$,and a user-defined function $\mathtt{d}$ we write $F' \in \mathscr{F}(\mathtt{o})$ and $F' \in \mathscr{F}(\mathtt{d})$ as short for

$$\mathscr{D}(\mathtt{o}) = \forall\overline{\alpha}\overline{\beta}.F \text{ and } F' = F[\overline{\alpha} := \overline{E}][\overline{\beta} := \overline{L}] \text{ for some closed expression types } \overline{E} \text{ and closed local types } \overline{L}$$

and

$$\mathscr{D}(\mathtt{d}) = \forall\overline{\alpha}\overline{\beta}.F \text{ and } F' = F[\overline{\alpha} := \overline{E}][\overline{\beta} := \overline{L}] \text{ for some closed expression types } \overline{E} \text{ and closed local types } \overline{L},$$

respectively.

A.1. Typing runtime expressions and configurations

The following clause extends to the auxiliary function *typeof*, defined in Fig. 6 (middle), to field values:

$$\mathit{typeof}(\{\delta_1 \mapsto \mathtt{l}_1\},\ldots,\{\delta_n \mapsto \mathtt{l}_n\}) = \mathtt{field}(L) \qquad \text{if } n \geq 1 \text{ and } L = \mathit{typeof}(\mathtt{l}_1) = \cdots = \mathit{typeof}(\mathtt{l}_n).$$

The predicate $\mathbf{valueHasType}(\delta,\Theta,v,E)$ checks whether the runtime value v is of type E w.r.t. the tree environment Θ on device δ. It is defined by the following clauses:

$$\begin{array}{rcll} \mathbf{valueHasType}(\delta,\Theta,\mathtt{l},L) & = & \mathbf{true}, & \text{if} \quad \mathit{typeof}(\mathtt{l}) = L \\ \mathbf{valueHasType}(\delta,\Theta,\phi,\Phi) & = & \mathbf{true}, & \text{if} \quad \mathbf{dom}(\phi) = \mathbf{dom}(\Theta),\delta \quad \text{and} \quad \mathit{typeof}(\phi) = \Phi \\ \mathbf{valueHasType}(\delta,\Theta,v,E) & = & \mathbf{false}, & \text{otherwise} \end{array}$$

Given a variable environment Γ and a type environment $\mathscr{X}$ we write $\mathbf{valueHasType}(\delta,\Theta,\Gamma,\mathscr{X})$ to mean that

$$\mathbf{dom}(\Gamma) = \mathbf{dom}(\mathscr{X}) \qquad \text{and} \qquad (\text{for all } \mathtt{x} \in \mathbf{dom}(\Gamma)) \quad \mathbf{valueHasType}(\delta,\Theta,\Gamma(\mathtt{x}),\mathscr{X}(\mathtt{x})).$$

We introduce three type systems for runtime expressions, for typing: initial configurations, final configurations, and (possibly non-initial and non-final) configurations, respectively (cf. Section 7.2).[7]

- The typing rules for final runtime expressions (judgement $\boxed{\delta_{\mathtt{uid}};\Theta;\mathscr{X} \vdash_{\mathbf{f}} a{\cdot}v : E}$) are reported in Figure 8.
- The typing rules for initial runtime expressions (judgement $\boxed{\delta_{\mathtt{uid}};\Theta;\mathscr{X} \vdash_{\mathbf{i}} a{\cdot}\circ : E}$) are reported in Figure 9.
- The typing rules for runtime expressions (judgement $\boxed{\delta_{\mathtt{uid}};\Theta;\mathscr{X} \vdash_{\mathbf{r}} e : E}$) are reported in Figure 10. Note that rules [R-FINAL] and [R-FUN-BODY] use the typing judgment for final runtime expressions in the premises, while rules [R-COND], [R-THEN], [R-ELSE], [R-FUN-ARG] use the typing judgment for initial runtime expressions in the premises.

Given a subexpression $\mathtt{e}$ of $\mathtt{e}_{\mathrm{main}}$ such that $\mathscr{X} \vdash \mathtt{e} : E$, the set of *well-formed tree environments for* $\mathtt{e}$ *in a device* δ, denoted by $\mathit{WFTE}(\delta,\mathtt{e})$, is inductively defined as follows:

$$(\delta_1 \mapsto e_1,\ldots,\delta_n \mapsto e_n) \in \mathit{WFTE}(\delta,\mathtt{e}) \ (n \geq 0) \text{ if } \delta \notin \{\delta_1,\ldots,\delta_n\} \text{ and for all } i \in 1..n$$

[7] Introducing specialized type systems for initial and final configurations simplifies the formulation of the type system for (possibly non-initial and non-final) configuration.

Final runtime-expression typing: $\boxed{\delta_{\mathtt{uid}};\Theta;\mathscr{X}\vdash_{\mathbf{f}} a{\cdot}v : E}$

$$\text{[F-VAR]}\ \frac{\textbf{valueHasType}(\delta,\Theta,v,E)}{\delta;\Theta;(\mathscr{X},\mathtt{x}:E)\vdash_{\mathbf{f}}\mathtt{x}{\cdot}v:E}\qquad \text{[F-LOCAL]}\ \frac{\textbf{valueHasType}(\delta,\Theta,\mathtt{l},L)}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}\mathtt{l}{\cdot}\mathtt{l}:L}$$

$$\text{[F-NBR]}\ \frac{\delta;\pi_{(\mathtt{nbr}\ [])}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e:L \qquad \textbf{valueHasType}(\delta,\Theta,\phi,\Phi)}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}(\mathtt{nbr}\ e){\cdot}\phi:\Phi}$$

$$\text{[F-THEN]}\ \frac{\begin{array}{l}\delta;\pi_{(\mathtt{if}\ []\ e_1\ \mathtt{e})}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} a{\cdot}t:\mathtt{bool}\\ \delta;\pi_{(\mathtt{if}\ a{\cdot}t\ []\ \mathtt{e})}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e_1:L \qquad \mathscr{X}\vdash \mathtt{e}:L\end{array}}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}(\mathtt{if}\ a{\cdot}t\ e_1\ \mathtt{e}){\cdot}\mathtt{l}:L}$$

$$\text{[F-ELSE]}\ \frac{\begin{array}{l}\delta;\pi_{(\mathtt{if}\ []\ \mathtt{e}\ e_2)}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} a{\cdot}f:\mathtt{bool}\\ \mathscr{X}\vdash \mathtt{e}:L \qquad \delta;\pi_{(\mathtt{if}\ a{\cdot}f\ \mathtt{e}\ [])}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e_2:L\end{array}}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}(\mathtt{if}\ a{\cdot}f\ \mathtt{e}\ e_2){\cdot}\mathtt{l}:L}$$

$$\text{[F-REP]}\ \frac{\mathscr{X}(\mathtt{w})=L \qquad \delta;\pi_{(\mathtt{rep}^{\mathtt{l}}\ \mathtt{x}\ \mathtt{w}\ [])}(\Theta);\mathscr{X}[\mathtt{x}:L]\vdash_{\mathbf{f}} e:L}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}(\mathtt{rep}^{\mathtt{l}}\ \mathtt{x}\ \mathtt{w}\ e){\cdot}\mathtt{l}:L}$$

$$\text{[F-OP]}\ \frac{\begin{array}{l}E_1\cdots E_n\to E\in\mathscr{F}(\mathtt{o})\\ \text{for all } i\in 1..n:\ \ \delta;\pi_{(\mathtt{o}\ e_1\ldots e_{i-1}\ []\ e_{i+1}\ldots e_n)}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e_i:E_i \qquad \textbf{valueHasType}(\delta,\Theta,v,E)\end{array}}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}(\mathtt{o}\ e_1\cdots e_n){\cdot}v:E}$$

$$\text{[F-FUN]}\ \frac{\begin{array}{l}(\mathtt{def}\ \mathtt{d}(\mathtt{x}_1\cdots\mathtt{x}_n)\ \mathtt{e}) \qquad E_1\cdots E_n\to E\in\mathscr{F}(\mathtt{d})\\ (\text{for all } i\in 1..n)\ \ \delta;\pi_{(\mathtt{d}^a\ e_1\ldots e_{i-1}\ []\ e_{i+1}\ldots e_n)}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e_i:E_i\\ \delta;\pi_{(\mathtt{d}^a[]\ e_1\cdots e_n)}(\Theta);\mathscr{X}[\mathtt{x}_1:E_1,\ldots\mathtt{x}_n:E_n]\vdash_{\mathbf{f}} a{\cdot}v:E \qquad \textbf{valueHasType}(\delta,\Theta,v,E)\end{array}}{\delta;\Theta;\mathscr{X}\vdash_{\mathbf{f}}(\mathtt{d}^a\ e_1\cdots e_n){\cdot}v:E}$$

Figure 8: Typing rules for final runtime expressions

- $|e_i| = \mathtt{e}$, and
- there exists $\Theta_i \in \mathit{WFTE}(\delta_i, \mathtt{e})$ such that $\delta_i;\Theta_i;\mathscr{X}\vdash_{\mathbf{f}} e_i : E$;

We say that a configuration $\langle\delta,\Theta,e\rangle$ such that $|e| = \mathtt{e}_{\mathsf{main}}$ and $\Theta\in\mathit{WFTE}(\delta,\mathtt{e}_{\mathsf{main}})$ is

- a *well-typed final configuration* if $\delta;\Theta;\bullet\vdash_{\mathbf{f}} e:L$;
- a *well-typed initial configuration* if $\delta;\Theta;\bullet\vdash_{\mathbf{i}} e:L$; and
- a *well-typed configuration* if $\delta;\Theta;\bullet\vdash_{\mathbf{r}} e:L$.

A.2. Auxiliary properties and proofs

The following lemma guarantees that the projection operator π (cf. Section 5.3) preserves well-formedness of tree environments.

Lemma A.1 (Projection π preserves well-formedness of tree environments). *Let* $\mathtt{e}$ *be a subexpression of* $\mathtt{e}_{main}$*. If* $\Theta\in\mathit{WFTE}(\delta,\mathtt{e})$*,* $|e|=\mathtt{e}$ *and* $e=(\mathbb{A}[e'])\cdot\hat{v}$ *then* $\pi_{\mathbb{A}}(\Theta)\in\mathit{WFTE}(\delta,|e'|)$*.*

Initial runtime expression typing: $\boxed{\delta_{\texttt{uid}};\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} a{\cdot}\circ : E}$

$$\frac{}{\delta;\ \Theta;\ (\mathscr{X}, \texttt{x} : E) \vdash_{\mathbf{i}} \texttt{x}{\cdot}\circ : E}\ [\text{I-VAR}] \qquad \frac{\textbf{valueHasType}(\delta, \Theta, \texttt{l}, L)}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} \texttt{l}{\cdot}\circ : L}\ [\text{I-LOCAL}]$$

$$\frac{\delta;\ \pi_{(\texttt{nbr}\ [])}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} a{\cdot}\circ : L}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\texttt{nbr}\ a{\cdot}\circ){\cdot}\circ : \Phi}\ [\text{I-NBR}]$$

$$\frac{\begin{array}{l}\delta;\ \pi_{(\texttt{if}\ []\ a_1{\cdot}\circ\ \texttt{e})}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} a{\cdot}\circ : \texttt{bool} \\ \delta;\ \pi_{(\texttt{if}\ a{\cdot}t\ []\ \texttt{e})}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} a_1{\cdot}\circ : L \qquad \mathscr{X} \vdash \texttt{e} : L\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\texttt{if}\ a{\cdot}\circ\ a_1{\cdot}\circ\ \texttt{e}){\cdot}\circ : L}\ [\text{I-THEN}]$$

$$\frac{\begin{array}{l}\delta;\ \pi_{(\texttt{if}\ []\ \texttt{e}\ a_2{\cdot}\circ)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} a{\cdot}\circ : \texttt{bool} \\ \mathscr{X} \vdash \texttt{e} : L \qquad \delta;\ \pi_{(\texttt{if}\ a{\cdot}f\ \texttt{e}\ [])}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} a_2{\cdot}\circ : L\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\texttt{if}\ a{\cdot}\circ\ \texttt{e}\ a_2{\cdot}\circ){\cdot}\circ : L}\ [\text{I-ELSE}]$$

$$\frac{\mathscr{X}(\texttt{w}) = L \qquad \delta;\ \pi_{(\mathring{\texttt{rep}}^{\mathring{\texttt{l}}}\ \texttt{x}\ \texttt{w}\ [])}(\Theta);\ \mathscr{X}[\texttt{x} : L] \vdash_{\mathbf{i}} a{\cdot}\circ : L}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\mathring{\texttt{rep}}^{\mathring{\texttt{l}}}\ \texttt{x}\ \texttt{w}\ a{\cdot}\circ){\cdot}\circ : L}\ [\text{I-REP}]$$

$$\frac{\begin{array}{l}E_1 \cdots E_n \to E \in \mathscr{F}(\texttt{o}) \\ \text{for all}\ \ i \in 1..n :\ \ \delta;\ \pi_{(\texttt{o}\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_i : E_i\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\texttt{o}\ e_1 \cdots e_n){\cdot}\circ : E}\ [\text{I-OP}]$$

$$\frac{\begin{array}{l}(\texttt{def}\ \texttt{d}(\texttt{x}_1 \cdots \texttt{x}_n)\ \texttt{e}) \qquad E_1 \cdots E_n \to E \in \mathscr{F}(\texttt{d}) \\ (\text{for all}\ i \in 1..n)\ \ \delta;\ \pi_{(\texttt{d}^{\circ}\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_i : E_i\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\texttt{d}^{\circ}\ e_1 \cdots e_n){\cdot}\circ : E}\ [\text{I-FUN-NULL}]$$

$$\frac{\begin{array}{l}(\texttt{def}\ \texttt{d}(\texttt{x}_1 \cdots \texttt{x}_n)\ \texttt{e}) \qquad E_1 \cdots E_n \to E \in \mathscr{F}(\texttt{d}) \\ (\text{for all}\ i \in 1..n)\ \ \delta;\ \pi_{(\texttt{d}^{a}\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_i : E_i \\ \delta;\ \pi_{(\texttt{d}[]\ e_1 \cdots e_n)}(\Theta);\ \mathscr{X}[\texttt{x}_1 : E_1, \ldots \texttt{x}_n : E_n] \vdash_{\mathbf{i}} a{\cdot}\circ : E\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{i}} (\texttt{d}^{a}\ e_1 \cdots e_n){\cdot}\circ : E}\ [\text{I-FUN}]$$

Figure 9: Typing rules for initial runtime expressions

Runtime expression typing: $\boxed{\delta_{\texttt{uid}};\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} \boldsymbol{e} : E}$

$$\frac{}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{f}} a{\cdot}v : E}\ \text{[R-FINAL]} \qquad \frac{}{\delta;\ \Theta;\ (\mathscr{X}, \mathtt{x} : E) \vdash_{\mathbf{r}} \mathtt{x}{\cdot}\circ : E}\ \text{[R-VAR]} \qquad \frac{\textbf{valueHasType}(\delta, \Theta, \mathtt{l}, L)}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} \mathtt{l}{\cdot}\circ : L}\ \text{[R-LOCAL]}$$

$$\frac{\delta;\ \pi_{(\mathtt{nbr}\ [])}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} e : L}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{nbr}\ e){\cdot}\circ : \Phi}\ \text{[R-NBR]}$$

$$\frac{\begin{array}{l}\delta;\ \pi_{(\mathtt{if}\ []\ e_1\ e_2)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} a{\cdot}\mathring{\mathtt{l}} : \mathtt{bool} \\ \delta;\ \pi_{(\mathtt{if}\ a{\cdot}t\ []\ e_2)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_1 : L \quad \delta;\ \pi_{(\mathtt{if}\ a{\cdot}f\ e_1\ [])}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_2 : L\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{if}\ a{\cdot}\mathring{\mathtt{l}}\ e_1\ e_2){\cdot}\circ : L}\ \text{[R-COND]}$$

$$\frac{\begin{array}{l}\delta;\ \pi_{(\mathtt{if}\ []\ e_1\ e_2)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} a{\cdot}t : \mathtt{bool} \\ \delta;\ \pi_{(\mathtt{if}\ a{\cdot}t\ []\ e_2)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} e_1 : L \quad \delta;\ \pi_{(\mathtt{if}\ a{\cdot}f\ e_1\ [])}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_2 : L\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{if}\ a{\cdot}t\ e_1\ e_2){\cdot}\circ : L}\ \text{[R-THEN]}$$

$$\frac{\begin{array}{l}\delta;\ \pi_{(\mathtt{if}\ []\ e_1\ e_2)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} a{\cdot}f : \mathtt{bool} \\ \delta;\ \pi_{(\mathtt{if}\ a{\cdot}t\ []\ e_2)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{i}} e_1 : L \quad \delta;\ \pi_{(\mathtt{if}\ a{\cdot}f\ e_1\ [])}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} e_2 : L\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{if}\ a{\cdot}f\ e_1\ e_2){\cdot}\circ : L}\ \text{[R-ELSE]}$$

$$\frac{\mathscr{X}(\mathtt{w}) = L \qquad \delta;\ \pi_{(\mathtt{rep}^{\mathring{\mathtt{l}}}\ \mathtt{x}\ \mathtt{w}\ [])}(\Theta);\ \mathscr{X}[\mathtt{x} : L] \vdash_{\mathbf{r}} a{\cdot}\circ : L}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{rep}^{\mathring{\mathtt{l}}}\ \mathtt{x}\ \mathtt{w}\ a{\cdot}\circ){\cdot}\circ : L}\ \text{[R-REP]}$$

$$\frac{\begin{array}{l}E_1 \cdots E_n \to E \in \mathscr{F}(\mathtt{o}) \\ \text{for all } i \in 1..n : \ \delta;\ \pi_{(\mathtt{o}\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} e_i : E_i\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{o}\ e_1 \cdots e_n){\cdot}\circ : E}\ \text{[R-OP]}$$

$$\frac{\begin{array}{l}(\mathtt{def}\ \mathtt{d}(\mathtt{x}_1 \cdots \mathtt{x}_n)\ \mathtt{e}) \qquad E_1 \cdots E_n \to E \in \mathscr{F}(\mathtt{d}) \\ (\text{for all } i \in 1..n)\ \ \delta;\ \pi_{(\mathtt{d}^\circ\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} e_i : E_i\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{d}^\circ\ e_1 \cdots e_n){\cdot}\circ : E}\ \text{[R-FUN-NULL]}$$

$$\frac{\begin{array}{l}(\mathtt{def}\ \mathtt{d}(\mathtt{x}_1 \cdots \mathtt{x}_n)\ \mathtt{e}) \qquad E_1 \cdots E_n \to E \in \mathscr{F}(\mathtt{d}) \\ (\text{for all } i \in 1..n)\ \ \delta;\ \pi_{(\mathtt{d}^a\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{r}} e_i : E_i \\ \delta;\ \pi_{(\mathtt{d}[]\ e_1 \cdots e_n)}(\Theta);\ \mathscr{X}[\mathtt{x}_1 : E_1, \ldots \mathtt{x}_n : E_n] \vdash_{\mathbf{i}} a{\cdot}\circ : E\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{d}^a\ e_1 \cdots e_n){\cdot}\circ : E}\ \text{[R-FUN-ARG]}$$

$$\frac{\begin{array}{l}(\mathtt{def}\ \mathtt{d}(\mathtt{x}_1 \cdots \mathtt{x}_n)\ \mathtt{e}) \qquad E_1 \cdots E_n \to E \in \mathscr{F}(\mathtt{d}) \\ (\text{for all } i \in 1..n)\ \ \delta;\ \pi_{(\mathtt{d}^a\ e_1 \ldots e_{i-1}\ []\ e_{i+1} \ldots e_n)}(\Theta);\ \mathscr{X} \vdash_{\mathbf{f}} e_i : E_i \\ \delta;\ \pi_{(\mathtt{d}[]\ e_1 \cdots e_n)}(\Theta);\ \mathscr{X}[\mathtt{x}_1 : E_1, \ldots \mathtt{x}_n : E_n] \vdash_{\mathbf{r}} a{\cdot}\circ : E\end{array}}{\delta;\ \Theta;\ \mathscr{X} \vdash_{\mathbf{r}} (\mathtt{d}^a\ e_1 \cdots e_n){\cdot}\circ : E}\ \text{[R-FUN-BODY]}$$

Figure 10: Typing rules for runtime expressions

PROOF. By induction on well-formed tree environments. The tree environment Θ must be of the form $(\delta_1 \mapsto e_1, \ldots, \delta_n \mapsto e_n)$ $(n \geq 0)$ with (for all $i \in 1..n$) $\delta \neq \delta_i$, $|e_i| = \mathtt{e}$ and (for some $\Theta_i \in WFTE(\delta_i, \mathtt{e})$) $\delta_i; \Theta_i; \mathscr{X} \vdash_{\mathbf{f}} e_i : E$.

Case $n = 0$**.** Immediate.

Case $n \geq 1$**.** Straightforward by induction, with a case analysis of the last rule used the derivations $\delta_i; \Theta_i; \mathscr{X} \vdash_{\mathbf{f}} e_i : E$. □

The following lemma guarantees that the initial configuration of a well-typed program (according to system $\vdash$ in Fig. 6) at the first round of computation is a well-typed initial configuration (according to system $\vdash_{\mathbf{i}}$).

Lemma A.2 (From surface to initial-runtime typing). *Let* $\mathtt{e}$ *be a subexpression of* $\mathtt{e}_{main}$*. If* $\mathscr{X} \vdash \mathtt{e} : E$ *and* $\Theta \in WFTE(\delta, \mathtt{e})$*, then* $\delta; \Theta; \mathscr{X} \vdash_{\boldsymbol{i}} \mathtt{e} : E$*.*

PROOF. By induction on a derivation of $\mathscr{X} \vdash \mathtt{e} : E$.

Cases [ST-VAR] and [ST-LOCAL]. Immediate by rules [I-VAR] and [I-LOCAL], respectively.

Case [ST-IF]. Straightforward by induction, using any of rules [I-THEN] and [I-ELSE].

Cases [ST-NBR], [ST-REP], [ST-OP] and [ST-FUN]. Straightforward by induction, using rules [I-NBR], [I-REP], [I-OP] and [I-FUN-NULL], respectively.

□

The following lemma guarantees that the initial configuration of a well-typed program in all the subsequent rounds of computation is a well-typed initial configuration (according to system $\vdash_{\mathbf{i}}$).

Lemma A.3 (From final- to initial-runtime typing). *Let* $\mathtt{e}$ *be a subexpression of* $\mathtt{e}_{main}$*. If* $\Theta, \Theta' \in WFTE(\delta, \mathtt{e})$*,* $|e| = \mathtt{e}$ *and* $\delta; \Theta; \mathscr{X} \vdash_{\boldsymbol{f}} e : E$ *then* $\delta; \Theta'; \mathscr{X} \vdash_{\boldsymbol{i}} init(e) : E$*.*

PROOF. By induction on a derivation of $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{f}} e : E$ (using Lemma A.1). First observe that the expression e must be of the form $a \cdot v$ (for some auxiliary rte a and value v) and recall that $init(a \cdot v)$ is the runtime expression obtained from $a \cdot v$ by dropping all annotations (not superscripts).

Cases [F-VAR] and [F-LOCAL]. Immediate by rules [I-VAR] and [I-LOCAL], respectively.

Cases [F-NBR], [F-THEN], [F-ELSE], [F-REP],[F-OP] and [F-FUN]. Straightforward by induction, using rules [I-NBR], [I-THEN], [I-ELSE], [I-REP], [I-OP] and [I-FUN], respectively.

□

The following lemma guarantees that a well-typed initial configuration (according to system $\vdash_{\mathbf{i}}$) is a well-typed configuration (according to system $\vdash_{\mathbf{r}}$).

Lemma A.4 (From initial-runtime to runtime typing). *Let* $\mathtt{e}$ *be a subexpression of* $\mathtt{e}_{main}$*. If* $\Theta \in WFTE(\delta, \mathtt{e})$*,* $|e| = \mathtt{e}$ *and* $\delta; \Theta; \mathscr{X} \vdash_{\boldsymbol{i}} e : E$ *then* $\delta; \Theta; \mathscr{X} \vdash_{\boldsymbol{r}} e : E$*.*

PROOF. By induction on a derivation of $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{i}} e : E$ (using Lemma A.1).

Cases [I-VAR] and [I-LOCAL]. Immediate by rules [R-VAR] and [R-LOCAL], respectively.

Cases [I-NBR], [I-THEN], [I-ELSE], [I-REP],[TI-OP], [I-FUN-NULL] and [I-FUN]. Straightforward by induction, using rules [R-NBR], [R-THEN], [R-ELSE], [R-REP], [R-OP], [R-FUN-NULL] and [R-FUN-ARG], respectively.

□

The following lemma guarantees that the domain of every field value occurring in a well-typed configuration consists of the identifiers of the aligned neighbours and of the identifier of the `uid` device.

Lemma A.5 (Domain alignment). *Let* $\mathtt{e}$ *be a subexpression of* $\mathtt{e}_{main}$. *Suppose* $\Theta \in WFTE(\delta, \mathtt{e})$, $|e| = \mathtt{e}$ *and* $\delta; \Theta; \mathscr{X} \vdash_f e : E$. *If* $e = a{\cdot}\phi$ *then* $\mathbf{dom}(\phi) = \mathbf{dom}(\Theta) \cup \{\delta\}$.

PROOF. By induction on a derivation of $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{f}} e : E$ (using Lemma A.1).

Cases [F-VAR] and [F-LOCAL]. Immediate.

Cases [F-NBR], [F-THEN], [F-ELSE], [F-REP],[F-OP] and [F-FUN]. Straightforward by induction.

□

The subject reduction theorem guarantees that reducing a well-typed configuration produces a well-typed configuration.

Theorem A.6 (Subject reduction). *Let* $\mathtt{e}$ *be a subexpression of* $\mathtt{e}_{main}$. *If* $\Theta \in WFTE(\delta, \mathtt{e})$, $|e| = \mathtt{e}$, $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : E$, $\mathbf{valueHasType}(\delta, \Theta, \Gamma, \mathscr{X})$ *and* $\delta; \Theta; \Gamma \vdash e \to e'$, *then* $|e'| = |e|$ *and* $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e' : E$.

PROOF. By induction on a derivation of $\delta; \Theta; \Gamma \vdash e \to e'$ with a case analysis of the reduction rule used.

Case [VAL]. $e = \mathtt{l}{\cdot}\circ$, $e' = \mathtt{l}{\cdot}\mathtt{l}$ and (by rule [R-LOCAL]) $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} \mathtt{l}{\cdot}\circ : L$. Then by rules [F-LOCAL] and [R-FINAL] we have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} \mathtt{l}{\cdot}\mathtt{l} : L$.

Case [VAR]. $e = \mathtt{x}{\cdot}\circ$, $e' = \mathtt{x}{\cdot}v$ (with $v = \Gamma(\mathtt{x})|_{\mathbf{dom}(\Theta),\delta}$) and (by rule [R-VAR]) $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} \mathtt{x}{\cdot}\circ : E$. Then by rules [F-VAR] and [R-FINAL] we have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} \mathtt{x}{\cdot}v : E$.

Case [NBR]. $e = (\mathtt{nbr}\ a{\cdot}\mathtt{l}){\cdot}\circ$, $e' = (\mathtt{nbr}\ a{\cdot}\mathtt{l}){\cdot}\phi$ (with $\phi = (\pi_{(\mathtt{nbr}\ [])}(\Theta), \delta \mapsto \mathtt{l})$) and (by rule [R-NBR]) $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : \Phi$. We have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{f}} a{\cdot}\mathtt{l} : L$ (which is the premise of rule [R-NBR]). Then by rules [F-NBR] and [R-FINAL] we have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e' : \Phi$.

Case [OP]. $e = (\mathtt{o}\ \overline{a}{\cdot}\overline{v}){\cdot}\circ$, $e' = (\mathtt{o}\ \overline{a}{\cdot}\overline{v}){\cdot}v$ (with $v = \varepsilon(\mathtt{o}, \overline{v})$) and (by rule [R-OP]) $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : E$. We have

$\mathscr{O}(\mathtt{d}) = E_1 \cdots E_n \to E$, and

for all $i \in 1..n$: $\delta; \pi_{(\mathtt{o}\ e_1 \ldots e_{i-1}\ [\,]\ e_{i+1} \ldots e_n)}(\Theta); \mathscr{X} \vdash_{\mathbf{f}} e_i : E_i$ and $e_i = a_i{\cdot}v_i$

(which are the premise of rule [R-OP]). Then by rules [F-OP] and [R-FINAL] we have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e' : E$.

Case [THEN]. $e = (\mathtt{if}\ a{\cdot}t\ a'{\cdot}\mathtt{l}\ e_2){\cdot}\circ$, $e' = (\mathtt{if}\ a{\cdot}t\ a'{\cdot}\mathtt{l}\ |e_2|){\cdot}\mathtt{l}$ and (by rule [R-THEN]) $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : L$. We have

$\delta; \pi_{(\mathtt{if}\ [\,]\ a'\mathtt{l}\ e_2)}(\Theta); \mathscr{X} \vdash_{\mathbf{f}} a{\cdot}t : \mathtt{bool}$,

$\delta; \pi_{(\mathtt{if}\ a{\cdot}t\ [\,]\ e_2)}(\Theta); \mathscr{X} \vdash_{\mathbf{f}} a'{\cdot}\mathtt{l} : L$, and

$\delta; \pi_{(\mathtt{if}\ a{\cdot}f\ a'\mathtt{l}\ [\,])}(\Theta); \mathscr{X} \vdash_{\mathbf{i}} e_2 : L$

(which are the premise of rule [R-THEN]). Since $|e_2|$ is a well-typed source expression, we have $\mathscr{X} \vdash |e_2| : L$. Then by rules [F-THEN] and [R-FINAL] we have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e' : L$.

Case [ELSE]. Similar to case [THEN] above.

Case [CONG]. By case analysis of the congruence context $\mathbb{C}$ used by rule [CONG] and of the typing rule used for deriving the judgement $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : E$.

Subcase $(\mathtt{nbr}\ [\,])$**.** Typing rule [R-NBR]. Straightforward by induction.

Subcase $(\mathtt{d}^{\mathtt{s}}\ \overline{e}\ [\,]\ \overline{e})$**.** Either typing rule [R-FUN-NULL] (if $\mathtt{s} = \circ$) or typing rule [R-FUN-ARG] (if $\mathtt{s} \neq \circ$). Both cases are straightforward by induction.

Subcase $(\mathtt{o}\ \overline{e}\ [\,]\ \overline{e})$**.** Typing rule [R-OP]. Straightforward by induction.

Subcase $(\mathtt{if}\ [\,]\ e\ e)$**.** Typing rule [R-COND]. Straightforward by induction.

Subcase $(\mathtt{if}\ a{\cdot}t\ [\,]\ e)$**.** Typing rule [R-THEN]. Straightforward by induction.

Subcase $(\texttt{if}\ \boldsymbol{a}\cdot\boldsymbol{f}\ \boldsymbol{e}\ [\,])$**.** Typing rule [R-ELSE]. Straightforward by induction.

Case [REP]. $e = (\texttt{rep}^{\mathring{\texttt{l}}_1}\ \texttt{x}\ \texttt{w}\ a)\cdot\circ$, $e' = (\texttt{rep}^{\mathring{\texttt{l}}_1 \triangleright \mathring{\texttt{l}}_2}\ \texttt{x}\ \texttt{w}\ a'\cdot\mathring{\texttt{l}}_2)\cdot\mathring{\texttt{l}}_2$ and (by rule [R-REP]) $\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e : L$. We have

$\mathscr{X}(\texttt{w}) = L$, and
$\delta;\pi_{(\texttt{rep}^{\mathring{\texttt{l}}_1}\ \texttt{x}\ \texttt{w}\ [])}(\Theta);\mathscr{X}[\texttt{x}:L]\vdash_{\mathbf{r}} a\cdot\circ : L$

(which are the premises of rule [R-REP]). By induction we have

$$\delta;\pi_{(\texttt{rep}^{\mathring{\texttt{l}}_1}\ \texttt{x}\ \texttt{w}\ [])}(\Theta);\mathscr{X}[\texttt{x}:L]\vdash_{\mathbf{r}} a'\cdot\mathring{\texttt{l}}_2 : L. \tag{1}$$

We have two subcases.

Subcase $\mathring{\texttt{l}}_2 = \circ$ Then, from (1) by rule [R-NBR], we have $\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e' : L$.

Subcase $\mathring{\texttt{l}}_2 = \texttt{l}_2$ Judgement (1) is derived by rule [R-FINAL] and we have
$\delta;\pi_{(\texttt{rep}^{\mathring{\texttt{l}}_1}\ \texttt{x}\ \texttt{w}\ [])}(\Theta);\mathscr{X}[\texttt{x}:L]\vdash_{\mathbf{f}} a'\cdot\texttt{l}_2 : L$ (which is the premise of rule [R-FINAL]). Then by rules [F-REP] and [R-FINAL] we have $\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e' : L$.

Case [FUN]. $e = (\texttt{d}^{\texttt{s}}\ \overline{a}\cdot\overline{v})\cdot\circ$, $e' = (\texttt{d}^{a}\ \overline{a}\cdot\overline{v})\cdot\mathring{v}$, $\mathscr{O}(\texttt{d}) = E_1\cdots E_n\to E$, $(\texttt{def}\ \texttt{d}(\texttt{x}_1\cdots\texttt{x}_n)\ \texttt{e}_0)$, and

$$\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e : E. \tag{2}$$

We have two subcases.

Subcase $\texttt{s} = \circ$ Judgement (2) is derived by rule [R-FUN-NULL]. We have

$$(\text{for all } i\in 1..n{:})\ \ \delta;\pi_{(\texttt{d}^{\circ}\ e_1\ldots e_{i-1}\ []\ e_{i+1}\ldots e_n)}(\Theta);\mathscr{X}\vdash_{\mathbf{r}} e_i : E_i \text{ where } e_i = a_i\cdot v_i \tag{3}$$

(which are premises of rule [R-FUN-NULL]) and

$$(\text{for all } i\in 1..n{:})\ \ \delta;\pi_{(\texttt{d}^{\circ}\ e_1\ldots e_{i-1}\ []\ e_{i+1}\ldots e_n)}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e_i : E_i \tag{4}$$

(since the premises of rule [R-FUN-NULL] must be derived by rule [R-FINAL]). Since $|e_0|$ is a well-typed source expression we have $\mathscr{X}\vdash|e_0| : E$ and (by Lemmas A.2 and A.4) $\delta;\Theta;\mathscr{X}[\texttt{x}_1:E_1,\ldots\texttt{x}_n:E_n]\vdash_{\mathbf{r}} e_0 : E$. By induction we have

$$\delta;\Theta;\mathscr{X}[\texttt{x}_1:E_1,\ldots\texttt{x}_n:E_n]\vdash_{\mathbf{r}} a\cdot\mathring{v} : E. \tag{5}$$

We have two subcases.

Subcase $\mathring{v} = \circ$ Then, from (3) and (5) by rule [R-FUN-BODY], we have $\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e' : L$.

Subcase $\mathring{v} = v$ Judgement (5) is derived by rule [R-FINAL] and we have
$\delta;\Theta;\mathscr{X}[\texttt{x}_1:E_1,\ldots\texttt{x}_n:E_n]\vdash_{\mathbf{f}} a\cdot v : E$. (which is the premise of rule [R-FINAL]). Then, from this judgement and (4), by rules [F-FUN] and [R-FINAL] we have $\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e' : L$.

Subcase $\texttt{s} = a_0$ Judgement (2) is derived by rule [R-FUN-BODY]. We have

$$(\text{for all } i\in 1..n{:})\ \ \delta;\pi_{(\texttt{d}^{\circ}\ e_1\ldots e_{i-1}\ []\ e_{i+1}\ldots e_n)}(\Theta);\mathscr{X}\vdash_{\mathbf{f}} e_i : E_i \text{ where } e_i = a_i\cdot v_i \tag{6}$$

$$\delta;\pi_{(\texttt{d}[]\ e_1\cdots e_n)}(\Theta);\mathscr{X}[\texttt{x}_1:E_1,\ldots\texttt{x}_n:E_n]\vdash_{\mathbf{r}} a_0\cdot\circ : E$$

(which are premises of rule [R-FUN-BODY]). By induction we have

$$\delta;\Theta;\mathscr{X}[\texttt{x}_1:E_1,\ldots\texttt{x}_n:E_n]\vdash_{\mathbf{r}} a\cdot\mathring{v} : E. \tag{7}$$

We have two subcases.

Subcase $\mathring{v} = \circ$ Then, from (6) and (7) by rule [R-FUN-BODY], we have $\delta;\Theta;\mathscr{X}\vdash_{\mathbf{r}} e' : L$.

Subcase $\mathring{v} = v$ Judgement (7) above must have been derived by rule [R-FINAL] and we have $\delta; \Theta; \mathscr{X}[\mathtt{x}_1 : E_1, \ldots \mathtt{x}_n : E_n] \vdash_{\mathbf{f}} a{\cdot}v : E$. (which is the premise of rule [R-FINAL]). Then from this judgement and (6) by rules [F-FUN] and [R-FINAL] we have $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e' : L$.

□

The progress theorem guarantees that if a non-final configuration is well typed then some reduction rule is applicable. We first prove the following auxiliary lemma.

Lemma A.7 (Progress). *Let $\mathtt{e}$ be a subexpression of $\mathtt{e}_{main}$. Suppose $\Theta \in WFTE(\delta, \mathtt{e})$, $|e| = \mathtt{e}$, $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : E$ and $\mathbf{valueHasType}(\delta, \Theta, \Gamma, \mathscr{X})$.*

1. *If $e = x{\circ}$, then $x \in \mathbf{dom}(\Gamma)$.*
2. *If $e = (\mathtt{nbr}\ a{\cdot}v){\cdot}\circ$ and $\pi_{(\mathtt{nbr}\ [])}(\Theta) = (\overline{\delta} \mapsto \overline{a}{\cdot}\overline{v})$, then $v\overline{v}$ are local values.*
3. *If $e = (\mathtt{if}\ a{\cdot}v\ e_1\ e_2){\cdot}\circ$, then v is a Boolean value.*
4. *If $e = (\mathtt{if}\ a{\cdot}t\ a'{\cdot}v\ e){\cdot}\circ$, then v is a local value.*
5. *If $e = (\mathtt{if}\ a{\cdot}f\ e\ a'{\cdot}v){\cdot}\circ$, then v is a local value.*
6. *If $e = (\mathtt{rep}^{\circ}\ x\ x'\ a){\cdot}\circ$, then $x' \in \mathbf{dom}(\Gamma)$.*
7. *If $e = (\mathtt{rep}^{\mathring{l}_1}\ x\ w\ a){\cdot}\circ$ and $\delta; \pi_{(\mathtt{rep}^{\mathring{l}_1}\ x\ w\ [])}(\Theta); \Gamma, (x := (\Gamma(w) \rhd \mathring{l}_1)) \vdash a \to a'{\cdot}v$, then v is a local value.*
8. *If $e = (o\ \overline{a}{\cdot}\overline{v}){\cdot}\circ$, then $exec(o, \overline{v})$ is defined.*
9. *If $e = (d^s\ \overline{a}{\cdot}\overline{v}){\cdot}\circ$, then $(\mathtt{def}\ d(\overline{x})\ \mathtt{e})$ and $\sharp(\overline{x}) = \sharp(\overline{v})$.*

PROOF.

1. Straightforward by inspecting rule [R-VAR].
2. Straightforward by inspecting rule [R-NBR], using the hypothesis $\Theta \in WFTE(\delta, \mathtt{e})$.
3. Straightforward by inspecting rules [R-COND], [R-THEN] and [R-ELSE].
4. Straightforward by inspecting rule [R-THEN].
5. Straightforward by inspecting rule [R-ELSE].
6. Straightforward by inspecting rule [R-REP], using the hypothesis $\mathbf{valueHasType}(\delta, \Theta, \Gamma, \mathscr{X})$.
7. Straightforward by inspecting rule [R-REP], using Theorem A.6.
8. Straightforward by inspecting rule [R-OP], using Lemma A.5 (recall that $\varepsilon(o, v_1, \cdots, v_n)$ is defined only if all the field values in $v_1, \ldots, v_n$ have the same domain, cf. end of Section 5.1).
9. Straightforward by inspecting rules [R-FUN-NULL], [R-FUN-ARG] and [R-FUN-BODY].

□

We can now prove the progress theorem.

Theorem A.8 (Progress). *Let $\mathtt{e}$ be a subexpression of $\mathtt{e}_{main}$. If $\Theta \in WFTE(\delta, \mathtt{e})$, $|e| = \mathtt{e}$, $\delta; \Theta; \mathscr{X} \vdash_{\mathbf{r}} e : E$ and $\mathbf{valueHasType}(\delta, \Theta, \Gamma, \mathscr{X})$, then either $e = a{\cdot}v$ (for some a and v) or $\delta; \Theta; \Gamma \vdash e \to$.*

PROOF. According to the rules in Figure 5, reduction gets stuck (i.e., it has reached a normal form that is not evaluated) only when:

1. In rule [VAR] a variable not in $\mathbf{dom}(\Gamma)$ is found.
2. In rule [NBR] an attempt to build an ill-formed field (i.e., a "field" that maps at least one device to a value that is not a local value) is made.
3. The condition of an `if`-expression evaluated to a not Boolean value.
4. In rule [THEN] the condition of the left branch of the `if`-expression evaluated to a field value.
5. In rule [ELSE] the condition of the right branch of the `if`-expression evaluated to a field value.
6. The second argument of a `rep`-expression is a variable that is not in $\mathbf{dom}(\Gamma)$.
7. The third argument of a `rep`-expression evaluated to a field value.
8. In rule [OP] the operator `o` is not defined on values $\overline{v}$ (this includes the case when the number of the values $\overline{v}$ is different from the number of arguments required by `o`).
9. The number of actual parameters of a function call is different from the number of formal parameters in the definition of the function.

Therefore the proof is straightforward by Lemma A.7.

□

We are now able to prove the domain-alignment and type-soundness theorems.

Restatement of Theorem 7.2 (Domain alignment). *For every well-typed program `P` and every device δ, if $\langle \delta, \Theta, e\rangle \in RC(\delta, P)$ and $\delta;\Theta;\bullet \vdash e \to e'$ (for some e') then every subreduction $\delta;\Theta_0;\Gamma \vdash e_1 \to e_2$ in the reduction $\delta;\Theta;\bullet \vdash e \to e'$ is such that if $e_2 = a{\cdot}\phi$ then $\boldsymbol{dom}(\phi) = \boldsymbol{dom}(\Theta_0) \cup \{\delta\}$.*

PROOF. Straightforward by Lemmas A.2, A.3 and A.5 and Theorem A.6.

□

Restatement of Theorem 7.3 (Type soundness). *For every well-typed program `P` and every device δ, if $\langle \delta, \Theta, e\rangle \in RC(\delta, P)$ and $\delta;\Theta;\bullet \vdash e \not\to$, then $\langle \delta, \Theta, e\rangle$ is evaluated.*

PROOF. Straightforward by Lemmas A.2, A.3, A.4 and Theorems A.6 and A.8.

□

References

[1] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *IEEE International Conference on Intelligent Robots and Systems (IROS '07)*, pages 2794–2800, 2007.

[2] J. Bachrach and J. Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT, March 2007.

[3] J. Bachrach, J. Beal, J. Horowitz, and D. Qumsiyeh. Empirical characterization of discretization error in gradient-based algorithms. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO) 2008*, October 2008.

[4] J. Beal. Engineered self-organization approaches to adaptive design. In R. Roy, E. Shehab, C. Hockley, and S. Khan, editors, *1st International Conference on Through-life Engineering Services*, pages 35–42. Cranfield University Press, November 2012.

[5] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.

[6] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975. ACM, 2008.

[7] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`.

[8] J. Beal, D. Pianini, and M. Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9), 2015.

[9] J. Beal and M. Viroli. Building blocks for aggregate programming of self-organising applications. In *Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, September 8-12, 2014*, pages 8–13, 2014.

[10] J. Beal and M. Viroli. Space–time programming. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 373(2046), 2015.

[11] J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.
[12] L. Birkedal, S. Debois, E. Elsborg, T. T. Hildebrandt, and H. Niss. Bigraphical models of context-aware systems. In L. Aceto and A. Ingólfsdóttir, editors, *Foundations of Software Science and Computation Structures, 9th International Conference, FOSSACS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25-31, 2006, Proceedings*, pages 187–201, 2006.
[13] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, MA, USA, 2002.
[14] L. Cardelli and P. Gardner. Processes in space. In *6th Conference on Computability in Europe*, volume 6158 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2010.
[15] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.
[16] S. S. Clark, J. Beal, and P. Pal. Distributed recovery for enterprise services. In *Proceedings of the IEEE Conference on Self-Adaptive and Self-Organising Systems 2015 (SASO 2015)*, 2015. In press.
[17] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982.
[18] F. Damiani, M. Viroli, D. Pianini, and J. Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *LNCS*, pages 113–128. Springer International, 2015.
[19] M. Egerstedt and X. Hu. Formation constrained multi-agent control. *IEEE Trans. on Robotics and Automation*, 17(6):947–951, 2001.
[20] N. Elhage and J. Beal. Laplacian-based consensus on spatial computers. In *AAMAS 2010*, 2010.
[21] B. R. Engstrom and P. R. Cappello. The sdef programming system. *Journal of Parallel and Distributed Computing*, 7(2):201 – 231, 1989.
[22] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.
[23] J. Fernandez-Marquez, G. Marzo Serugendo, S. Montagna, M. Viroli, and J. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, 12(1):43–67, 2013.
[24] M. M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, Oct. 2008.
[25] D. Helbing and A. Johansson. Pedestrian, crowd and evacuation dynamics. In *Encyclopedia of Complexity and Systems Science*, pages 6476–6495. Springer, 2009.
[26] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.
[27] C. Lasser, J. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.
[28] M. Mamei, R. Menezes, R. Tolksdorf, and F. Zambonelli. Case studies for self-organization in computer science. *Journal of Systems Architecture*, 52(8-9):443–460, 2006.
[29] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.
[30] A. Mansutti, M. Miculan, and M. Peressotti. Multi-agent systems design and prototyping with bigraphical reactive systems. In K. Magoutis and P. Pietzuch, editors, *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 201–208, 2014.
[31] R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation*, 204(1):60 – 122, 2006.
[32] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.
[33] MIT Proto. software available at `http://proto.bbn.com/`, Retrieved January 1st, 2012.
[34] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, pages 78–87, Aug. 2004.
[35] R. Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *IEEE Trans. on Automatic Control*, 51(3), March 2006.
[36] A. Omicini and M. Viroli. Coordination models and languages: From parallel computing to self-organisation. *The Knowledge Engineering Review*, 26(1):53–59, Mar. 2011. Special Issue 01 (25th Anniversary Issue).
[37] G. Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108 – 143, 2000.
[38] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 2013.
[39] D. Pianini, M. Viroli, and J. Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853, 2015.
[40] D. Pianini, M. Viroli, F. Zambonelli, and A. Ferscha. Hpc from a self-organisation perspective: The case of crowd steering at the urban scale. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pages 460–467, July 2014.
[41] F. Raimbault and D. Lavenier. Relacs for systolic programming. In *Int'l Conf. on Application-Specific Array Processors*, pages 132–135, October 1993.
[42] J.-J. Slotine and W. Wang. A study of synchronization and group cooperation using partial contraction theory. *Cooperative Control*, pages 443–446, 2005.
[43] M. Viroli. Engineering confluent computational fields: from functions to rewrite rules. In *Spatial Computing Workshop (SCW 2013)*, AAMAS 2013, Saint Paul, Minnesota, USA, May 2013.
[44] M. Viroli, J. Beal, and M. Casadei. Core operational semantics of Proto. In *Proceedings of ACM SAC 2011*, pages 1325–1332. ACM, 21–25 Mar. 2011.
[45] M. Viroli, J. Beal, F. Damiani, and D. Pianini. Efficient engineering of complex self-organising systems by self-stabilising fields. In *Proceedings of the IEEE Conference on Self-Adaptive and Self-Organising Systems 2015 (SASO 2015)*, 2015. In press.
[46] M. Viroli, J. Beal, and K. Usbeck. Operational semantics of proto. *Science of Computer Programming*, 78(6):633–656, June 2013.
[47] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces.

ACM Transactions on Autonomous and Adaptive Systems, 6(2):14:1 – 14:24, June 2011.

[48] M. Viroli, M. Casadei, and A. Omicini. A framework for modelling and implementing self-organising coordination. In *Proceedings of ACM SAC 2009*, volume III, pages 1353–1360. ACM, 8–12 Mar. 2009.

[49] M. Viroli, F. Damiani, and J. Beal. A calculus of computational fields. In C. Canal and M. Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.

[50] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Proceedings of Coordination 2012*, volume 7274 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2012.

[51] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In S. Ossowski, P. Lecca, C.-C. Hung, and J. Hong, editors, *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, pages 295–302, Riva del Garda, TN, Italy, 26-30 March 2012. ACM.

[52] M. Viroli, D. Pianini, S. Montagna, G. Stevenson, and F. Zambonelli. A coordination model of pervasive service ecosystems. *Science of Computer Programming*, 110:3 – 22, 2015.

[53] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, Mar. 2004.

[54] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.

[55] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.

[56] L. Xiao, S. Boyd, and S. Lall. A scheme for asynchronuous distributed sensor fusion based on average consensus. In *Fourth International Symposium on Information Processing in Sensor Networks*, 2005.

[57] C.-H. Yu and R. Nagpal. Self-adapting modular robotics: A generalized distributed consensus framework. In *International Conference on Robotics and Automation (ICRA)*, 2009.

[58] F. Zambonelli, G. Castelli, L. Ferrari, M. Mamei, A. Rosi, G. D. M. Serugendo, M. Risoldi, A.-E. Tchao, S. Dobson, G. Stevenson, J. Ye, E. Nardini, A. Omicini, S. Montagna, M. Viroli, A. Ferscha, S. Maschek, and B. Wally. Self-aware pervasive service ecosystems. *Procedia CS*, 7:197–199, 2011.

[59] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.